

# *Dicas* C e Assembly para arquitetura x86-64



Frederico Lamberti Pissarra



# **Dicas - C e Assembly para arquitetura x86-64**

Versão 0.33.9

© 2014-2017 por Frederico Lamberti Pissarra  
3 de fevereiro de 2017

**Título:** C & Assembly para arquitetura x86-64

**Autor:** Frederico Lamberti Pissarra

**Ano de publicação:** 2016

Este material é protegido pela licença GFDL 1.3:

*C & Assembly para arquitetura x86-64*

*Copyright (C) 2014-2016 by Frederico Lamberti Pissarra*

A permissão de cópia, distribuição e/ou modificação deste documento é garantida sob os termos da licença “GNU Free Documentation License, Version 1.3” ou versão mais recente publicada pela Free Software Foundation.

O texto integral da licença pode ser lido no link

<https://www.gnu.org/licenses/fdl.html>.

É importante, no entanto, advertir ao leitor que este material encontra-se em processo de produção. Trata-se, então, de um rascunho e não reflete o material finalizado.

Figura da capa: *DIE da arquitetura Intel Ivy Bridge*

# Índice

<b>Introdução</b>	<b>1</b>
Organização do livro	2
Sistemas Operacionais: Linux vs Windows	2
Sobre C e C++	2
Sobre a linguagem Assembly	3
A “melhor” rotina possível	4
Códigos-fonte deste livro	5
Arquiteturas de processadores Intel	6
Não confunda arquitetura com modo de operação	7
Avisos finais sobre o livro	7
<b>Capítulo 1: Introdução ao processador</b>	<b>9</b>
Modos de operação	9
O que significa “proteção”?	10
O que significa “paginação”?	10
Proteção e segmentos, no modo protegido em 32 bits	10
Onde fica a tabela de descritores?	13
Seletores e descritores de segmentos no modo x86-64	13
Descritores no modo x86-64	14
Cache de descritores	15
Proteção no modo x86-64	15
<b>Capítulo 2: Interrompemos nossa programação...</b>	<b>17</b>
O que é uma interrupção?	17
As Interrupções especiais: Exceções, Faltas e Abortos	17
Existem dois tipos de interrupções de hardware diferentes	18
Interrupções são sempre executadas no kernelspace!	18
Faltas importantes para performance	19
Sinais: Interrupções no user space	19
Um exemplo de uso de sinais, no Linux	21
Não use signal(), use sigaction()	21
Existe mais sobre sinais do que diz sua vã filosofia...	22
<b>Capítulo 3: Resolvendo dúvidas frequentes sobre a Linguagem C</b>	<b>23</b>
Headers e Módulos: Uma questão de organização	23
Chamadas de funções contidas no mesmo módulo	24
Tamanho de inteiros e ponteiros	25
Não é prudente confiar nos tamanhos dos tipos default	25
Existe diferença entre “declarar” e “definir” um símbolo	26
Diferença entre declaração e uso	27
Parênteses	27
Escopo dos parâmetros de funções	27
Protótipos de funções	28
Problemas com algumas estruturas com máscaras de bits	29
Finalmente: Ponteiros!	31
Declarando e usando ponteiros	32
Problemas com ponteiros para estruturas	34
Ponteiros e strings	34
Diferenças entre declarar ponteiros e arrays contendo “strings”	35

Mais ponteiros e arrays	36
Declarando e inicializando arrays, estruturas e unions	36
Ponteiros, funções e a pilha	38
Entendendo algumas declarações “malucas” usando ponteiros	39
Utilidade de ponteiros para funções	40
A biblioteca padrão: libc	41
A libc faz um monte de coisas “por baixo dos panos”	43
Cuidados ao usar funções da libc	44
Disparando processos filhos	44
<b>Capítulo 4: Resolvendo dúvidas sobre a linguagem Assembly</b>	<b>47</b>
O processador pode ter “bugs”	47
A mesma instrução pode gastar mais tempo do que deveria	48
Nem todas as instruções “gastam” tempo	49
A pilha e a “red zone”	49
Prefixos	50
<b>Capítulo 5: Misturando C e Assembly</b>	<b>53</b>
Convenções de chamada (x86-64)	53
Convenções de chamada (i386)	54
Funções com número de parâmetros variável no x86-64	56
Pilha, nos modos i386 e x86-64	56
Um detalhe sobre o uso de registradores de 32 e 64 bits	57
Exemplo de função em assembly usando a convenção de chamada	57
Aviso sobre retorno de funções com tipos complexos	58
O uso da pilha	60
Variáveis locais e a pilha	61
O compilador não usa todas as instruções	62
Detalhes interessantes sobre a instrução NOP e o prefixo REP	63
LOOP e LOOPNZ são diferentes, mas REPNZ e REP são a mesma coisa!	65
Assembly inline	65
Operador de indireção (ponteiros) em assembly	68
Usando a sintaxe Intel no assembly inline do GCC	69
Problemas com o assembler inline do GCC	69
Usando NASM	70
<b>Capítulo 6: Ferramentas</b>	<b>73</b>
GNU Linker (ld)	73
Obtendo informações com objdump	75
GNU Debugger	75
Configurando o GDB	76
Duas maneiras de listar código assembly no gdb	76
Listando registradores	77
Examinando a memória com o GDB	78
Obtendo listagens em assembly usando o GCC	78
Sobre os endereços em listagens em assembly obtidas com objdump ou GCC	81
<i>Usando o make: Fazendo um bolo, usando receitas</i>	82
<b>Capítulo 7: Medindo performance</b>	<b>87</b>
Ciclos de máquina e ciclos de clock	87
Contar ciclos de máquina não é fácil	87
Como medir?	88

Aumentando a precisão da medida	89
Mas, o gcc possui funções “intrínsecas” para executar CPUID e RDTSC!	91
Melhorando a medição de performance	92
O cálculo do ganho de performance	92
Quando um ganho “vale à pena”?	93
Usando “perf” para medir performance	94
<b>Capítulo 8: Otimizações “automáticas”</b>	<b>95</b>
Níveis de otimização	95
“Common Subexpression Elimination” (CSE)	95
Desenrolamento de loops	96
Movendo código invariante de dentro de loops (Loop Invariant Code Motion)	96
Eliminação de código morto (Dead Code Elimination)	97
Eliminação de armazenamento morto (Dead Store Elimination)	97
Previsão de saltos (Branch Prediction)	97
Simplificações lógicas	99
Simplificação de funções recursivas	100
Auto vetorização (SSE)	101
Otimizações através de profiling	101
<b>Capítulo 9: Caches</b>	<b>103</b>
A estrutura dos caches	103
Determinando o tamanho de uma linha	104
O dilema do alinhamento	105
Dica para usar melhor os caches...	106
Audaciosamente indo onde um byte jamais esteve...	107
Funções inline e a saturação do cache	107
<b>Capítulo 10: Memória Virtual</b>	<b>109</b>
Virtualização de memória	109
Espaços de endereçamento	109
Paginação	109
Paginação e swapping	110
Tabelas de paginação no modo x86-64	111
As entradas da Page Table	112
As extensões PAE e PSE	112
Translation Lookaside Buffers	113
A importância de conhecer o esquema de paginação	113
Tabelas de páginas usadas pelo userspace	114
Alocando memória: malloc e mmap	115
Um exemplo de injeção de código, usando páginas	116
Quanta memória física está disponível?	118
<b>Capítulo 11: Threads!</b>	<b>121</b>
Multitarefa preemptiva e cooperativa	121
Múltiplos processadores	122
Como o scheduler é chamado?	123
Finalmente, uma explicação de porque SS não é zero no modo x86-64!	123
Na prática, o que é uma “thread”?	123
Criando sua própria thread usando pthreads	124
Criando threads no Windows	125
Parar uma thread “na marra” quase sempre não é uma boa ideia	127

Nem todas as threads podem ser “mortas”	127
Threads, núcleos e caches	128
Trabalhar com threads não é tão simples quanto parece	128
Evitando “race conditions”	129
Threads e bibliotecas	130
Threads e Bloqueios	131
O que significa isso tudo?	131
Tentando evitar o chaveamento de contextos de tarefas	131
Usando o OpenMP	132
OpenMP não é mágico	133
Compilando e usando OpenMP	133
OpenCL e nVidia CUDA	134
<b>Capítulo 12: Ponto flutuante</b>	<b>135</b>
Precisão versus Exatidão	135
O que é “ponto flutuante”?	135
Estrutura de um float	135
Analogia com “notação científica”	136
Valores especiais na estrutura de um float.	137
Pra que os valores denormalizados existem?	138
Intervalos entre valores	138
Um problema de base	139
O conceito de valor “significativo”	140
Regras da matemática elementar nem sempre são válidas	141
Que tal trabalhar na base decimal, ao invés da binária?	142
A precisão decimal de um tipo float ou double	142
Comparando valores em ponto flutuante	143
Não compare tipos de ponto flutuante diferentes	144
Evite overflows!	144
Ponto fixo	145
Modo x86-64 e ponto flutuante	146
O tipo long double	146
<b>Capítulo 13: Instruções Estendidas</b>	<b>149</b>
SSE	149
Funções “intrínsecas” para SSE.	150
Exemplo do produto escalar	151
Uma “otimização” que falhou – o produto vetorial	152
Uma otimização bem sucedida: Multiplicação de matrizes	154
E quando ao AVX?	155
Outras extensões úteis: BMI e FMA	155
<b>Capítulo 14: Dicas e macetes</b>	<b>159</b>
Valores booleanos: Algumas dicas interessantes em C	159
Quanto mais as coisas mudam...	160
INC e DEC são lerdas!	160
Não faça isso!!!	160
Aritmética inteira de multipla precisão	161
Você já não está cansado disso?	162
Otimização de preenchimento de arrays	164
Tentando otimizar o RFC 1071 check sum. E falhando...	166
Previsão de saltos e um array de valores aleatórios	168

Usar raiz quadrada, via SSE, parece estranho...	170
Gerando números aleatórios	171
<b>Capítulo 15: Misturando Java e C</b>	<b>173</b>
<i>Porque não é uma boa ideia misturar C com ambientes “gerenciados”</i>	173
Garbage Collection, no Java	174
Misturando com um exemplo simples...	175
Usando strings	178
Usando arrays unidimensionais	180
Usando objetos	181
Chamando métodos do próprio objeto	182
Acessando membros de dados do próprio objeto	184
De volta aos arrays: Usando mais de uma dimensão	184
<b>Capítulo 16: Usando Python como script engine</b>	<b>187</b>
Contagem de referências	187
Instanciando o python	187
Carregando um módulo	188
Executando um módulo simples	189
<b>Apêndice A: System calls</b>	<b>191</b>
Considerações sobre o uso da instrução SYSCALL	191
Onde obter a lista de syscalls do Linux?	192
Usar syscalls no Windows não é uma boa idéia!	193
<b>Apêndice B: Desenvolvendo para Windows usando GCC</b>	<b>195</b>
Usando o MinGW no Linux	195
Usando o MinGW no Windows	195
As bibliotecas mingwm10.dll e msrvct.dll	196
Limitações do MinGW	196
Assembly com o MinGW e NASM	196
Windows usa codificação de caracteres de 16 bits, internamente	196
Vale a pena desenvolver aplicações inteiras em assembly para Windows?	197
Importando DLLs no MinGW-w64	198
<b>Apêndice C: Built-ins do GCC</b>	<b>201</b>
<b>Apêndice D: Módulos do Kernel</b>	<b>205</b>
Anatomia de um módulo simples	205



# Introdução

Em 1994 publiquei uma série de 26 capítulos sobre linguagem assembly numa rede de troca de mensagens conhecida como RBT. Na época a Internet ainda era restrita ao círculo acadêmico e inacessível a nós, meros mortais. Era também uma época em que o processador “top de linha” não passava do 286 e, só na segunda metade dos anos 90, tive acesso a um 386DX (40 MHz), onde comecei a explorar os novos registradores e instruções em 32 bits. O “curso de assembly”, como ficou conhecido, para minha surpresa, fez um tremendo sucesso depois que a RBT deixou de existir, sendo citado até hoje em fóruns especializados e copiado entre estudantes universitários (alguns me relataram) país à fora. O “curso” me colocou em contato com muita gente e, assim me dizem, é tido como uma espécie de manual definitivo (embora não seja!). Acontece que tudo fica velho e obsoleto. Algumas das dicas e truques contidos no “curso” não valem mais ou não são mais usados em lugar nenhum, mesmo pelos melhores compiladores. Por exemplo, hoje é meio difícil acessar os recursos de placas de vídeo diretamente e ninguém mais usa os device drivers HIMEM.SYS e EMM386.EXE.

Durante anos pensei em atualizar o “curso”. Só que não queria publicar algo para o completo novato ou repetir a estrutura do texto original. Não quero ensinar C e Assembly pra ninguém. Existem bons manuais, livros e tutoriais disponíveis por ai, tanto em forma textual quanto em vídeo, no Youtube, basta fazer uma pesquisa no Google! Ao invés disso, este livro é uma coletânea de artigos que escrevi em alguns blogs e sites especializados desde a época do “curso” – e com algumas novidades. É uma coleção de dicas e macetes, bem como a tentativa de “mastigar” um pouco alguns conceitos que podem ser obtidos facilmente no *Wikipedia*, só que numa linguagem tecnicamente mais escabrosa para o estudante. É o caso de capítulos que falam de cache, “memória virtual” e threads, por exemplo.

Mesmo que a minha intenção de facilitar o entendimento sobre tópicos tão áridos seja bem sucedida, devo alertá-lo para alguns detalhes: Primeiro, este material não é “básico”. Algum entendimento prévio sobre as linguagens C e Assembly é necessário. Segundo, a intenção **não** é esmiuçar as entranhas de um sistema operacional ou de seu ambiente de janelas preferido. Você verá muita informação **resumida** sobre interrupções, gerenciamento de memória, threads, mas o assunto abordado aqui está longe de ser completo. Um bom começo para mergulhar nesses assuntos são os manuais de desenvolvimento de software da Intel<sup>1</sup> e da AMD<sup>2</sup>. Mas, essas não são as únicas referências bibliográficas importantes. Você poderá querer ler um bocado sobre chipsets (legados, por exemplo, como o PIC 8259A<sup>3</sup> e o PIT 8253<sup>4</sup>, ou os mais modernos, com o I/O APIC 82093A<sup>5</sup>), bem como listagens dos mapeamentos de portas de I/O<sup>6</sup>. Esse material é essencial para o entendimento da arquitetura dos PCs, especialmente no que concerne o mapeamento da memória.

Já para a linguagem C existem, além de tutoriais online, bons livros traduzidos para o português, bem como as especificações ISO da linguagem (que recomendo fortemente que você se familiarize!).

---

1 “IA-32 & Intel 64 Software Development Manuals”: <http://goo.gl/UbssTF>.

2 “AMD64 Architecture Programmer’s Manuals”: <http://goo.gl/oIBnVD>.

3 Programmable Interrupt Controller 8259A Datasheet: <https://goo.gl/jCu2sy>.

4 Programmable Interval Timer 8253 Datasheet: <http://goo.gl/9zGsvF>.

5 Advanced I/O Programmable Interrupt Controller 82093A Datasheet: <http://goo.gl/44ntD7>.

6 Eis uma listagem, em formato texto: <http://goo.gl/cF82AV>.

## **Organização do livro**

Os capítulos 1 e 2 dão o embasamento para entendimento sobre recursos do processador. Nos capítulos 3 e 4 mostro algumas dicas sobre C e Assembly, respectivamente, para tentar resolver, espero que definitivamente, dúvidas que frequentemente surgem aos meus leitores e amigos. Sigo falando sobre como “misturar” códigos em C e Assembly. Falo também alguma coisa sobre a mistura com Java, no capítulo 15, e Python, no capítulo 16. Mesmo isto não sendo o foco desse material.

O capítulo 6 fala alguma coisa sobre as “ferramentas do ofício” para o desenvolvedor. Não é um material extensivo, só uma introdução.

Os capítulo 7 e 8 começam a mostrar alguma coisa sobre performance. O capítulo 7 é especialmente interessante: Ele mostra uma maneira de medir a performance de suas rotinas. E você encontrará, no apêndice B, duas funções escritas em assembly, que possibilitam uma medição mais “precisa”.

A partir do nono capítulo até o décimo terceiro são mostrados recursos importantes para performance, existentes no seu processador. O entendimento sobre caches, memória virtual e threads é particularmente importante se você quer arrancar até o último ciclo de máquina de performance em suas aplicações.

No capítulo 12 esqueço um pouco sobre performance e exploro alguns conceitos sobre operações em ponto flutuante, no sentido de mostrar que nem tudo é o que parece. O 13º capítulo estende o 12º, mostrando as extensões SSE, AVX e FMA.

O antepenúltimo capítulo é dedicado a dicas em geral.

E, no final, os apêndices dão informações adicionais que podem ser úteis para consulta e uso.

## **Sistemas Operacionais: Linux vs Windows**

Na maioria do tempo falarei sobre Linux aqui. Isso não significa que dicas e macetes não possam ser aplicadas a outros sistemas, como Windows e OS/X. Acontece que o único sistema operacional onde podemos colocar as mãos em suas entradas e realizar uma avaliação quase que cirúrgica é o Linux<sup>7</sup>.

A não ser que você tenha o código fonte do Windows mais recente ou do OS/X (que é baseado no FreeBSD, por falar nisso), certos recursos não podem ser conhecidos diretamente. Quaisquer explicações envolvendo recursos internos do Windows, por exemplo, seria meramente especulativa. A documentação existente sobre a API é boa (disponível na *MSDN Library*<sup>8</sup>, por exemplo), mas é superficial do ponto de vista das entradas do sistema. Acredito que usar a imaginação com base em informações vindas de especulação é algo que pode ser feito na interpretação de contos de fadas, não num material técnico...

Na medida do possível mostrarei dicas que envolvem também o Windows. Embora o OS/X também seja proprietário, é mais fácil encontrar paralelos com o sistema que lhe deu origem (FreeBSD). Assim, dentre os sistemas operacionais mais “famosos” este livro é restrito a apenas dois: Linux e Windows, com ênfase no primeiro.

## **Sobre C e C++**

De acordo com o título deste livro, este material também é restrito a duas linguagens de programação: C e Assembly. As poucas referências a C++, Java e C# estarão lá apenas para ilustrar

---

<sup>7</sup> Ok, você também pode fazer isso com as variações “free” do BSD!

<sup>8</sup> Acessível via <https://msdn.microsoft.com/library>.

algum ponto. As únicas exceções são os capítulos sobre a “mistura” de Java e Python com C. É meio difícil falar sobre “misturar” código C com Java sem falar em Java, né?

De maneira geral, não lido aqui com C++ e linguagens mais “moderninhas”, ou orientadas à objetos por questões de convicção. Em essência, elas são a mesma coisa que C. Têm algumas coisinhas a mais (classes, sobrecarga de operadores, funções virtuais, etc) que podem ser implementadas cuidadosamente por um programador com boa experiência em C. Outro motivo para não falar sobre C++ é que ele é mais complicado de entender em baixo nível. O leitor que quiser usar assembly junto ao seu código C++ deve prestar atenção e tomar cuidado com muitos detalhes. Não são poucos! Para citar algumas:

- **Name Mangling** – Os nomes de funções, em C++, costumam ser codificados de uma maneira diferente do que é feito em C. Em C uma função do tipo: “int f(int);” tem o nome de “f”, numa listagem do código equivalente em assembly. Em C++, o nome é uma coisa maluca como “\_Z1fi” ou algo que o valha (não vou me ater à convenção de nomenclatura de funções em C++ aqui... isso muda de compilador para compilador). O verbo “To mangle”, em inglês, pode ser literalmente traduzido para “mutilar”. O que diz muito sobre C++;
- **Funções membro de classes** – As funções membro, não estáticas, de uma classe recebem um ponteiro adicional, escondido, chamado ‘this’. Este ponteiro aponta para a “instância” do objeto da classe e é passado, escondido, para todas as funções (de novo: não estáticas).
- **Funções membro virtuais** – Toda chamada a uma função membro virtual é feita com dupla indireção. Isto é, uma referência a um objeto é um ponteiro que aponta para uma tabela contendo ponteiros para as funções. Explicarei isso, brevemente, no capítulo sobre “misturas”;

Existem mais algumas diferenças que devem ser estudadas por quem quer usar assembly com C++ e pretendo dedicar um capítulo só para te mostrar porque acredito que orientação à objetos **não** é uma boa ideia... Corro o risco de criar uma guerra entre os entusiastas, já que afirmo que um programa escrito em C costuma ser mais performático e mais simples do que um escrito em C++ devido a menor complexidade do código gerado pelo compilador, no segundo caso. Esses são, em essência, os meus motivos pela preferência ao C, em vez de C++.

Essas considerações não querem dizer que C++ seja uma perfeita droga. Ao contrário: O compilador C++ faz um excelente trabalho de otimização, assim como o compilador C. Mas, existem complexidades e abstrações, que facilitam a vida do programador e, ao mesmo tempo, causam a criação de código menos que ótimo – do ponto de vista de código equivalente em C e Assembly otimizados. Isso quer dizer que o desenvolvedor C++ deve ter preocupação redobrada se quiser códigos de excelente performance... O desenvolvedor C também tem que fazê-lo, mas a preocupação é menor...

## Sobre a linguagem Assembly

Quanto a linguagem assembly, existem muitos “sabores” que o leitor pode escolher. Cada tipo de processador tem o seu. Este texto lida apenas com os modos i386 e x86-64 dos processadores da família Intel. Não lido aqui com as plataformas IA-64 (também da Intel), ou ARM e ARM-64, por exemplo (o último está “em voga”, hoje em dia, graças aos dispositivos móveis e devices como o *Raspberry PI* e smartphones). Isso é importante porque nessas outras arquiteturas temos, além da linguagem assembly diferente, regras de otimizações e organização de hardware também completamente diferentes.

Assembly, aqui, também **não** é usada como linguagem principal de desenvolvimento e recomendo **fortemente** que você não tente usá-la como tal. Hoje em dia acho uma verdadeira loucura

desenvolver aplicações inteiras nessa linguagem. Até os sistemas operacionais são complexos o suficiente para não permitirem uma abordagem produtiva nesse sentido e, além do mais, os compiladores de nível maior, como é o caso de C, fazem um trabalho muito bom do ponto de vista da otimização de código.

É bom notar que os únicos compiladores com grande flexibilidade de otimizações são, necessariamente, C e C++. Isso é particularmente válido com o uso de compiladores do projeto GNU: *gcc* e *g++*<sup>9</sup>. Compiladores de outras linguagens como C#, Java, Pascal... possuem algumas poucas opções de otimizações (geralmente apenas uma opção do tipo “-optimize”), mas não a mesma flexibilidade. Outros compiladores C/C++ já foram bons, um dia, como é o caso da variação contida no Visual Studio. Hoje, a maioria das opções de otimização foram extirpadas para tornarem o compilador mais “amigável” à plataforma .NET e sua *Intermediate Language*. Dessa forma, mesmo para Windows, continuo recomendando o GCC... Uma outra possibilidade é o *clang*, projeto *open source* criado pela Apple. O *clang* é, essencialmente, a mesma coisa que o GCC com algumas *features* extras que, do ponto de vista do desenvolvedor em linguagem C, são inócuas, na maioria das vezes.

Claro que o trabalho de otimização pode ser feito pelo desenvolvedor esperto puramente em assembly, mas o mesmo desenvolvedor pode dar um tiro no próprio pé com sua “esperteza” e criar código cuja performance será pior do que a gerada pelo compilador C. Já tive experiências dolorosas nesse sentido. O lema “Não existe melhor otimizador do que aquele que está entre suas orelhas”<sup>10</sup> continua válido e, em alguns casos, assembly é a escolha definitiva para atingir o objetivo de ter um código rápido, só que nem sempre isso funciona bem! Na maioria das vezes recomendo confiança (adotando precauções) no trabalho do compilador aliado ao projeto cuidadoso de algoritmos, inclusive com um bom apoio da literatura extensa sobre o assunto. Um bom algoritmo, muitas vezes, cria bons códigos finais e dispensa otimizações mais *hardcore*. Não deixe de ler *The Art of Computer Programming* de Donald E. Knuth e *Algorithms in C* de Robert Sedgewick<sup>11</sup>.

Mesmo achando uma coisa de maluco desenvolver aplicações inteiras em Assembly, não posso deixar de dizer que o conhecimento dessa linguagem e da arquitetura de seu processador (e do PC) são essenciais para atingir alta performance. Observar o que seu compilador preferido gerou e melhorar o código de forma que você tenha certeza de que conseguiu o melhor resultado só é possível ao analisá-lo ao nível do Assembly. E, às vezes, criar uma ou outra rotina crítica diretamente em Assembly poupa tempo e torna seu código mais simples, além de rápido.

Outro aviso sobre esse livro é que certos conceitos fundamentais da linguagem são assumidos como conhecidos. Uma discussão mais profunda sobre Assembly pode ser encontrada num outro livro: *Linguagem Assembly para i386 e x86-64*, deste mesmo autor que vos escreve, ó leitor!

## A “melhor” rotina possível

Alguns leitores me perguntam, de tempos em tempos, qual é o “melhor” jeito de fazer alguma coisa. Entendo que por “melhor” querem dizer “o mais veloz”, no sentido de que exetem o mais rápido possível. Bem... Não existe tal coisa... O que existe é um conjunto de fatores que levarão a sua rotina a ser “mais rápida” do que outra equivalente e isso só pode ser determinado de três formas:

1. Experiência;
2. Experimentação;

---

<sup>9</sup> O compilador do Visual Studio teve grande parte de suas opções de otimização retiradas da linha de comando. Nesse sentido, ele **não** gera o melhor código possível.

<sup>10</sup> Li isso num livro de Michael Abrash: *Zen of Code Optimization*.

<sup>11</sup> Sedgewick também tem publicado um excelente livro chamado somente de *Algorithms*, onde todos os exemplos são em Java. Deixando Java de lado, vale muito a pena lê-lo.

### 3. Medição.

Por “experiência” quero dizer a familiaridade do desenvolvedor com a linguagem e com o ambiente. Somente essa convivência poderá te dizer qual “jeito” que criará rotinas mais rápidas ou mais lentas... A “experimentação” é essencial, até mesmo para o sucesso do primeiro item. Já “medição” é sempre um passo necessário. Usar uma técnica que sempre deu certo no passado não é garantia de que ela dará certo hoje. É necessário medir a performance de vários casos para determinar qual é o mais rápido. Mais adiante neste livro apresentarei uma rotina simples para medir a quantidade de “ciclos de clock” gastos por uma rotina. É essencial que essas medições sejam feitas em códigos críticos, mesmo que você ache que sua rotina seja a “mais rápida possível”...

### Códigos-fonte deste livro

Todo código-fonte completo que você ler neste texto começa com uma linha de comentário dizendo o nome do arquivo que usei. Com isso você pode copiá-lo e compilá-lo. Essa convenção é particularmente importante quando temos códigos que possuem diversos módulos ou arquivos.

Se preferir, alguns dos códigos mais importantes estão disponíveis no *GitHub*, no endereço <http://github.com/fredericopissarra/book-srcs>.

No decorrer do livro chamo de “módulos” cada um dos arquivos com extensão “.c” ou “.asm”, ou seus arquivos objetos equivalentes. Um programa completo pode ser composto de diversos módulos que são compilados separadamente e depois *linkados* para formar o arquivo executável. Também é conveniente que você aprenda alguma coisa a respeito do utilitário “make”, isto é, como construir um *makefile*. Um *makefile* é uma receita que o utilitário *make* usa para compilar e linkar todos os módulos do seu programa e construir o produto final... Mais adiante dou uma “palinha” sobre o *make*, só pra dar um gostinho da coisa.

**Avisos:** Sobre *makefiles*: Os comandos de uma “receita” têm que ser precedidos de um caractere ‘t’ (tab). Nas listagens, neste livro, esse caractere não está lá. Ao copiar e colar você obterá erros ao chamar o ‘make’.

Sobre as listagens em C e Assembly: O editor de textos usado para confeccionar esse livro substitui as aspas por dois caracteres especiais... A aspa de abertura é diferente da aspa de fechamento. Observe: “”. Ao copiar o código e tentar compilá-lo, obterá erros, com toda certeza... Tentei acertar esse problema nos códigos e acredito que consegui. Mas, esteja avisado desse possível problema!

Não uso IDEs para desenvolver minhas aplicações, apenas o bom e velho *vim*, os compiladores e muito material de referência (manuais, livros, *manpages* etc). Este é um dos motivos porque você não verá screenshots de janelas, receitas de configuração de meu “ambiente de desenvolvimento” favorito, etc. Este livro é sobre desenvolvimento em C usando Assembly, para o modo x86-64 do processador Intel, com uma pitada de informações sobre hardware e sobre algumas entradas do processador ou do PC. Ele **não** é um livro sobre IDEs.

Já que não uso IDEs e elas me dão alergia (assim como o Windows!), nas listagens com vários módulos, por motivos de brevidade, indicarei o início do novo arquivo através de uma linha que lembra um “picote”, em modo texto:

-----%<----- corte aqui -----%<-----

Assim, você saberá onde termina um arquivo e começa outro. O exemplo abaixo mostra um módulo em C chamado “hello.c”, um *makefile* (chamado de “Makefile”) e a linha de comando usada para compilar o projeto:

```

/* hello.c */
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello, world!\n");
    return 0;
}
-----%<---- corte aqui -----%<-----
# Makefile

hello: hello.o
    $(CC) -o $@ $^

hello.o: hello.c
    $(CC) -O3 -c -o $@ $<
-----%<---- corte aqui -----%<-----
$ make
cc -O3 -c -o hello.o hello.c
cc -o hello hello.o
$ ./hello
Hello, world!

```

No caso de linhas de comando, os comandos “digitáveis” aparecerão em negrito. E, já que meu foco é Linux, o “prompt” é sempre um '\$' ou '#' (se o nível de acesso exige o usuário 'root'). Você poderá ver um prompt “C:>” para linhas de comando específicas para Windows, mas será raro.

Outro recurso, no que se refere aos códigos fonte, é o uso de reticências para dizer que não estou mostrando um código completo, mas um fragmento. Isso indica que mais alguma coisa deve ser feita na listagem para que ela seja compilável, mas isso não é interessante no momento, já que quero mostrar apenas a parte significativa do código. Um exemplo é o caso onde posso mostrar um loop usando uma variável que deve ser inicializada em algum outro lugar, só que para efeitos de explicação, isso é supérfluo:

```

/* Essas reticências dizem que mais alguma coisa é necessária aqui! */
...
/*
Estamos, no momento, interessados apenas neste loop!
Óbviamente os ponteiros 'dp' e 'sp' foram inicializados em algum outro lugar! */
while (*dp++ = *sp++);
/*
E aqui tem mais código...
...

```

Tento, também, manter os códigos fonte contidos numa única página ou separado entre páginas de uma maneira menos fragmentada possível, para facilitar a leitura.

Você também reparou que comentários tem cor deferente, certo?

## **Arquiteturas de processadores Intel**

O termo “arquitetura” é usado de forma bastante ampla neste livro. Existem dois conceitos fundamentais: Quando os termos i386<sup>12</sup> (ou IA-32) e x86-64 (ou “Intel 64”) são usados, refiro-me aos processadores derivados da família 80x86 que suportam e operam nos modos de 32 e 64 bits, respectivamente. O outro conceito está atrelado à tecnologia usada no processador. Existem várias e elas têm nomes engraçados: Nehalem, NetBurst, Sandy Bridge, Ivy Bridge, Haswell etc, para citar apenas os da Intel.

---

<sup>12</sup> Usarei i386 ao invés de IA-32 aqui, mesmo que i386 seja específico para uma arquitetura de processadores (80386SX e 80386DX).

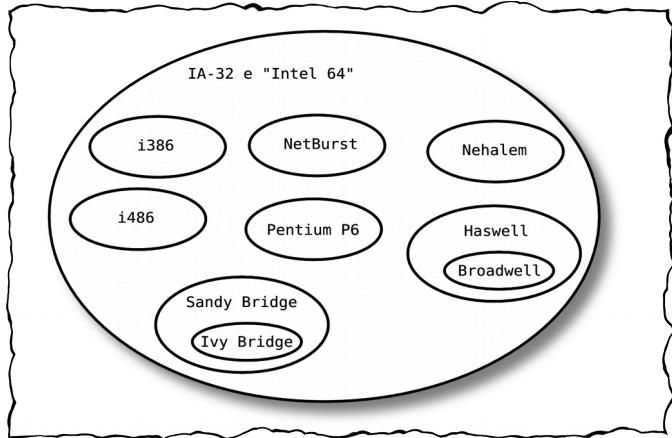


Figura 1: Arquiteturas dos processadores Intel

Saber sobre essas arquiteturas é útil, por exemplo, porque foi apenas na *NetBurst* que a Intel introduziu o modo de 64 bits.

Em essência, este livro aplica-se a todas as tecnologias, desde o i386 até a mais recente (Broadwell, na época em que escrevo isso), mas meu foco está mais nas arquiteturas vindas depois da *NetBurst*, especialmente na *Haswell*. Meus ambientes de teste atuais são baseados nessa última arquitetura.

Não é estranho que um livro que se propõe a falar sobre arquitetura de 64 bits recorra a tecnologias de 32? Acontece que o modo x86-64 é, na realidade, o mesmo modo i386 com extensões de 64 bits e com alguns recursos extirpados. Quase tudo o que vale para 32 bits, vale para 64. O contrário é que não se aplica...

Assim, quando falo da arquitetura i386, estou falando de todos os processadores Intel depois do i386, inclusive. Quando falo de x86-64, todos os processadores depois da arquitetura NetBurst, inclusive, são o objeto de estudo.

## **Não confunda arquitetura com modo de operação**

Neste livro, o termo i386 aplica-se tanto a arquitetura quanto a um modo de operação dos processadores da família x86.

Quando ler “modo i386”, isso significa que o processador estará trabalhando em modo “protegido” de 32 bits. O outro modo de operação citado na documentação da Intel é “IA-32e”. Este é o modo de operação de 64 bits ou, de maneira mais precisa, o modo de 32 bits com extensões (daí o ‘e’) para 64 bits. Neste livro chamo esse modo de “x86-64”, aproveitando a nomenclatura da AMD. Acho “x86-64” mais sexy do que “IA-32e”. O modo x86-64 é também chamado de “amd64”, este último é usado na nomeação de pacotes do Linux.

No texto você poderá ver uma mistura de nomenclaturas nesse sentido... IA-32e, amd64 e x86-64 são usados de forma intercambiável para significar a mesma coisa: o modo de 64 bits. Pode ser que você encontre IA-32 e i386 também, que querem dizer a mesma coisa.

## **Avisos finais sobre o livro**

Todo esse livro fala somente sobre a unidade central de processamento (CPU) ou “processador”, para os íntimos. Mostrarei muito pouco (se mostrar!), ou apenas um resumo, sobre GPUs (*Graphical Processing Unit*), chipsets e outros dispositivos contidos no seu computador.

Você também vai reparar que uso o termo “plataforma Intel” no texto, mesmo sabendo que a AMD é um grande competidor, a Intel firmou-se como padrão *de facto* com relação a esse tipo de processador... Isso não significa que a AMD fique atrás. Aliás, recomendo que você estude **também**

os manuais de desenvolvimento de software da AMD (que são mais “mastigáveis”).

Só mais uma coisa que você vai reparar... Uso a notação KiB, MiB, GiB, TiB e PiB para “quilo”, “mega” “giga”, “tera” e “peta” bytes ao invés das tradicionais “kB”, “MB” etc. O motivo é simples: Os prefixos “k”, “M”, “G”, “T” e “P” são potências de 10, enquanto os padrões ISO/IEC 80000-13 e IEEE 1541-2002 padronizam os prefixos “kibi”, “mebi”, “gibi”, “tebi” e “pebi” como sendo multiplicadores na base 2, respectivamente  $2^{10}$ ,  $2^{20}$ ,  $2^{30}$ ,  $2^{40}$  e  $2^{50}$ .

Ainda, o “B”, maiúsculo indica “byte” e o minúsculo, “b”, indicará “bit”, como em 15 Mib/s (15 mebibits por segundo).

Ao invés de “kibi” continuarei chamando de “quilo” só para não causar maior estranheza ou gozação (“é de comer”??).

# Capítulo 1: Introdução ao processador

Quem pretende desenvolver software para a plataforma Intel não deveria estar interessado em um modelo de processador específico (i7, i5, i3, Core2 Duo, Pentium, 486, 386 etc), mas nos pontos comuns entre eles. No entanto, quanto mais você mergulhar em detalhes, verá que não existe tal coisa de “arquitetura comum”. A cada par de anos a Intel apresenta uma nova arquitetura nomeada a partir de algum lugar ou cidade norte-americana (Sandy Bridge, Ivy Bridge, Haswell etc) que possui uma série de novos recursos. A arquitetura *Haswell*, por exemplo, possui quase o dobro de poder de processamento que sua irmã mais nova, a arquitetura *Sandy Bridge*, sem contar com algumas novidades.

Sem levar em conta a arquitetura, o que pretendo mostrar neste capítulo são informações que possam ser usadas para atingir a programação de alta performance na plataforma x86-64, se o leitor estiver preocupado com isso... A ênfase, como será em todo o livro, é o “modo protegido” de 64 bits.

## Modos de operação

Processadores da família Intel 80x86, a partir do 80286, podem trabalhar em diversos modos. Só estamos interessados nos processadores que suportem o modo x86-64. Este é o modo protegido, paginado, **de 32 bits com extensões para 64 bits**. É importante perceber que não existe um “modo de 64 bits”, mas uma extensão ao modo de 32. Note, também, os termos “protegido” e “paginado”. Uma breve explicação sobre proteção e paginação é dada nos tópicos seguintes.

Existem outros modos de operação do processador que são interessantes e usados, em casos muito específicos, mas que não explorarei aqui. Por exemplo, o processador inicia sua vida (depois do *reset* ou *power up*) no modo *real*. Esse é um modo de 16 bits, onde o processador só enxerga 1 MiB de memória e um monte de recursos mais complicados, do modo protegido, são simplificados. É o modo usado pelo antigo MS-DOS e pelo Windows 3.1... A vida era simples e boa naqueles tempos. Hoje ela ficou complicada e cheia de necessidades. Por isso, o modo real é usado apenas pela BIOS e por uma parcela ínfima do *bootstrap* do seu sistema operacional.

Sim! A BIOS opera em modo real de 16 bits! E é por isso que sistemas operacionais como Linux e Windows **não a usam nunca!** Do mesmo jeito, um programinha escrito para MS-DOS não funcionará nesses sistemas diretamente. Para executá-los, se você ainda tiver alguma peça de museu dessas, terá que usar um emulador como o *DOSBox*, uma máquina virtual com o MS-DOS instalado ou um modo de operação especial do processador chamado *Virtual8086*.

Outro modo menos conhecido é o SMM (*System Management Mode*). Para o desenvolvedor de aplicações esse modo é irrelevante, já que é dedicado para sistemas operacionais e, mesmo assim, raramente usado, de acordo com minhas observações. Você poderá achar mais informações sobre SMM nos manuais da Intel.

Os únicos modos que nos interessam serão chamados, neste livro, de i386 e x86-64. O primeiro é o tradicional modo protegido de 32 bits e o segundo é o modo estendido de 64. Esse “estendido” é importante porque existem dois modos x86-64: O *compatible* e o *long*. O primeiro (*compatible*) é, essencialmente, o modo i386 com suporte aos registradores de 64 bits. Todo o resto funciona do mesmo jeito que no modo i386. Já o segundo modo (*long*) funciona como um modo completamente novo, onde algumas características “malucas” do modo i386 não existem. A maioria dos sistemas operacionais ditos de 64 bits, baseados em processadores Intel, da família 80x86, usam o modo

“long” e, portanto, não falarei do modo “compatible” aqui.

O modo “long” é o que chamo aqui de modo x86-64.

## **O que significa “proteção”?**

Num ambiente onde vários processos podem estar sendo executados de forma concorrente é importante que um não possa interferir no outro. Por “interferência” podemos entender que um processo não pode ter acesso aos dados e código de outro. Pelo menos não diretamente.

Proteção, neste sentido, é a infraestrutura oferecida pelo processador para isolarmos processos. Através da proteção podemos, por exemplo, ter códigos e dados do kernel completamente isolados de códigos e dados de um programa do usuário em execução e, ainda, um programa do usuário não tem como acessar recursos de outro programa. Um não “sabe” que o outro existe. Assim, cada processo tem o seu próprio “espaço” de memória, protegido.

Veremos que apenas o kernel, ou o sistema operacional, têm como acessar todo e qualquer recurso de processos, mesmo que não sejam os seus próprios. Mas o contrário não vale.

## **O que significa “paginação”?**

Ao invés do processador trabalhar apenas com a memória física, contida nos seus pentes de memória, enxergando-a como um grande array, a paginação permite dividí-la em blocos pequenos chamados de “páginas”. Mais do que isso: Paginação nos permite mapear mais memória do que a que existe, fisicamente, fazendo um malabarismo que descreverei lá no capítulo sobre “memória virtual”.

Através do sistema de paginação temos dois “espaços” de memória<sup>13</sup>: O **espaço físico** e o **espaço linear** (ou Virtual). O espaço físico é aquele onde um endereço é um índice que especifica diretamente na memória. Já um espaço linear está relacionado a uma outra forma de endereçamento: Neste espaço, um endereço linear é um valor que precisa ser traduzido para um endereço físico. Um endereço linear especifica uma entrada num conjunto de tabelas que contém descritores de páginas. Este endereço, linear, também contém o deslocamento dentro de uma página. Ou seja, um endereço linear é uma maneira de endereçamento indireta.

Qual é a utilidade de “quebrarmos” o espaço físico em páginas? É que no esquema de tradução de um endereço linear para um endereço físico temos que usar “mapas” ou “tabelas” que indicam se uma página está presente na memória física ou não e onde, no espaço físico ela está. Se a página não existir fisicamente, o sistema operacional tem a opção de trocar (*swap*) uma página existente por uma não existente, aproveitando o espaço físico e fazendo de conta que temos mais memória do que está instalada no seu sistema. Essas “trocas” geralmente são feitas entre o espaço de memória física ocupada pela “página” e algum lugar no disco (HD). No caso do Linux é comum que uma partição de swap seja usada para esse fim. No caso do Windows, um arquivo de swap geralmente é usado.

A paginação é um modo opcional no modo i386, mas é obrigatório no modo x86-64.

## **Proteção e segmentos, no modo protegido em 32 bits**

Se você já estudou assembly para a família 80x86, já viu que existem seis registradores “seletores de segmentos” (CS, DS, ES, SS, FS e GS). No modo “real”, de 16 bits, o conteúdo desses registradores fornece um endereço base de 20 bits que aponta para um “segmento” de 64 KiB de tamanho. Isso é feito deslocando o conteúdo de um desses registradores para a esquerda em 4 bits,

---

13 Na verdade são 3: O espaço Lógico, o Virtual e o Físico.

que é a mesma coisa que multiplicar o valor por 16, efetivamente adicionando esses bits extras (zerados) à direita do valor contido nesses registradores. De posse desse endereço base adicionamos um deslocamento e obtemos um **endereço lógico** de 20 bits:

$$Endr = (\text{Segmento} \cdot 16) + \text{Deslocamento}$$

No modo protegido a coisa é um pouco mais complicada. Esses registradores **não** contém o endereço base de um segmento. Eles contém um índice que **seleciona** uma entrada em uma **tabela de descritores**.

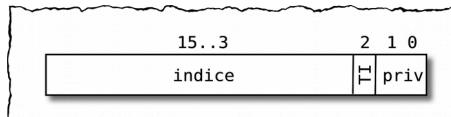


Figura 2: Estrutura de um selector de segmentos.

O diagrama acima mostra a estrutura de um seletor. Temos 13 bits de índice, 1 bit que nos diz o “tipo” de seletor (que não importa agora) e dois bits que nos dizem o “privilegio” do seletor (que explico mais adiante). Com 13 bits, um seletor pode selecionar uma das 8192 entradas da tabela de descritores.

A *tabela de descritores* contém entradas que **descrevem** (daí o nome!) segmentos de memória que podem ser selecionados pelo **seletor** (de novo, daí o nome!). Cada entrada nessa tabela é chamada de **descritor** e têm mais ou menos a seguinte estrutura simplificada<sup>14</sup>:

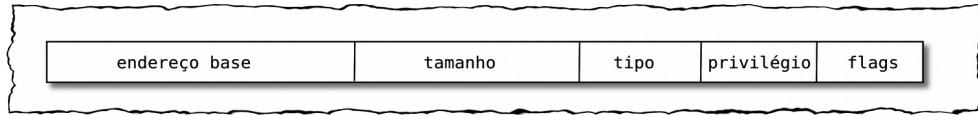


Figura 3: Forma resumida de um descritor.

No modo i386, sem usarmos o recurso de paginação, um bloco (segmento) de memória descrito num **descritor** pode ter até 4 GiB (depende do campo “tamanho” do descritor) e estar localizado em qualquer lugar da memória física (de acordo com o campo “endereço base”). O campo “tipo” nos diz para que finalidade o segmento de memória será usado (dados, código, pilha e outras estruturas usadas pelo processador). O campo “flags” também têm essa finalidade.

O campo “privilegio” contém um valor que indica qual é o nível de acesso que um processo precisa ter para usar esse descritor. Esse campo “privilegio”, num descritor, é chamado de DPL (*Descriptor Priviledge Level*). Não confundir com o campo “privilegio” num **seletor**: Lá ele é chamado de RPL (*Requestor Priviledge Level – Nível de privilégio do requisitante*). O DPL diz ao processador qual o privilégio necessário para acessar o segmento e o RPL nos diz qual é o privilégio que o seletor está requisitando.

Só que ambos os privilégios precisam ser comparados com o nível de privilégio corrente (CPL, ou *Current Priviledge Level*). CPL é o privilégio com o qual o processador está trabalhando no momento e é mantido no seletor de segmento de código (registrador CS). O CPL também é mantido, como cópia, no registrador SS (*Stack Selector*) porque o processo exige o uso da pilha em instruções como CALL e RET, bem como no tratamento de interrupções... Digo cópia porque o campo de privilégio do seletor SS é, de fato, RPL, mas precisa ter o mesmo nível do CPL.

Eis um exemplo dos CPL, DPL e RPL em uso: Suponha que a seguinte instrução esteja prestes a ser executada:

```
mov eax, [0x400104]
```

<sup>14</sup> A estrutura não é assim. Este é um esquema para facilitar a compreensão.

Suponhamos que o CPL seja 3 (O seletor de código, CS, aponta para algum lugar do *userspace*) e suponha também que o RPL do seletor DS seja, também, 3. Este RPL (de DS) será checado contra o DPL na entrada da tabela de descritores correspondente ao índice contido no seletor e, contra o CPL. No caso, se o DPL também for 3, então o processador poderá executar a instrução, caso contrário ele causará um *segmentation fault* ou um *general protection error*.

Neste caso, todos os três níveis de privilégio (CPL, RPL e DPL) precisam ser 3, já que este é menor privilégio possível. Um código executado (CPL) no privilégio 3 não pode requisitar, via seletor de dados (RPL de DS) acesso a um descritor com privilégio maior (se DPL for menor que 3).

A mesma coisa acontece com a referência ao endereço linear formado pelo endereço base contido na tabela de descritores, de acordo com o índice em DS, e o offset 0x400104, fornecido na instrução: Se o RPL de DS, o DPL do descritor e o CPL não baterem, teremos também um *segmentation fault* ou GPL. Por “baterem” eu não quero dizer que precisem ser iguais. Se o CPL for 0, basicamente os privilégios não serão checados, já que o processador, executando instruções neste nível, terá acesso total a qualquer coisa.

Como esses “privilégios” têm apenas 2 bits de tamanho, eles podem variar entre 0 e 3, onde o nível de privilégio 0 é o mais poderoso e o 3 o menos. A regra é que privilégios com valores menores são mais privilegiados. Um processo rodando com CPL 2 pode acessar recursos descritos com DPL 2 ou 3, acessados via seletores com RPL 2 ou 3 (mas um seletor com RPL 3 não pode usar um descritor com DPL 2!). A ordem do poder do privilégio pode, inicialmente, parecer estranha, mas pense numa corrida: Quem chega em 1º lugar é o vencedor. Os que chegam em 2º ou 3º, por definição, são perdedores. Como a contagem começa de zero, privilégios desse valor são sempre mais “poderosos”.

O nível de privilégio 0, por ser mais poderoso, permite a execução de todas as instruções do processador. Já os privilégios menores tem limitações significativas. A execução de algumas instruções “privilegiadas” também causará *segmentation fault*. Se tentarmos executar uma instrução LGDT, por exemplo, e estivermos no *userspace*, teremos problemas... Geralmente, o kernel de um sistema operacional e seus módulos, “rodam” no nível de privilégio zero (*system* ou *kernel mode*). Os programas criados por você rodarão no nível 3 (*user mode*). A figura abaixo mostra uma possibilidade de distribuição de privilégios. Na prática apenas os níveis 0 e 3 são usados por causa do sistema de paginação (memória virtual), que discutirei mais tarde.

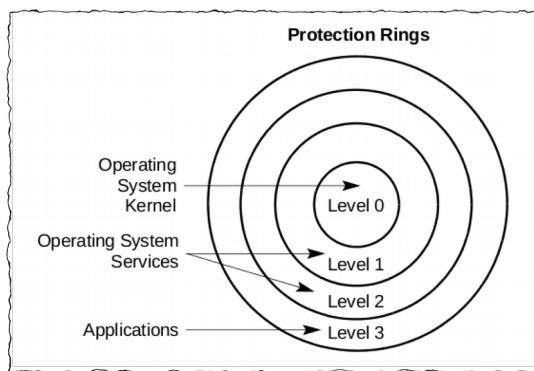


Figura 4: "Anéis" de privilégio.

Outra nomenclatura que você pode achar para códigos e dados que estejam presentes na região de memória descrita com nível de privilégio zero é *kenelland* (a “terra do kernel”) ou *kernelspace*. Se estiverem no privilégio 3 pertencerão ao *userland* (a “terra do usuário”) ou *userspace*.

## Onde fica a tabela de descritores?

O processador não possui uma região de memória interna que mantém a tabela de descritores. Essa tabela tem que ser colocada na memória física. Cada descritor, no modo i386, ocupa exatamente 8 bytes e, como temos 8192 descritores (lembre-se dos 13 bits do índice num seletor), então uma tabela ocupa exatamente 64 KiB na memória física.

Quando o modo protegido é inicializado, carregamos o endereço inicial da tabela de descritores *globais* num registrador especial chamado GDTR (*Global Descriptor Table Register*)<sup>15</sup>. O processador usa esse endereço como base para localizar as entradas na tabela de descritores, de acordo com o índice contido num seletor. A primeira entrada da tabela é sempre zerada e considerada inválida. Esse é um descritor NULO<sup>16</sup>. Um índice zero num seletor apontará para esse descritor e, qualquer acesso à memória usando esse seletor causará um *segmentation fault* ou GPF (*General Protection Fault*). As demais entradas da tabela descrevem regiões da memória linear para uso específico... A configuração depende do kernel. O sistema operacional pode, por exemplo, usar a entrada de índice 1 para descrever um segmento de código com DPL igual a zero, o índice 2 para um segmento de dados e o índice 3 para um segmento de pilha, ambos com o mesmo privilégio, no *kernelspace*. Todos esses 3 descritores, óbviamente, estarão disponíveis apenas para o kernel (que usa CPL e RPL zerados nos seletores). Outras entradas na tabela poderão descrever segmentos com DPL=3 que poderão ser executados, lidos e gravados através de seletores com RPL ou CPL=3 (ou CPL com “maior” privilégio que 3).

Outras tabelas, como as tabelas de páginas, descritas no capítulo sobre “memória virtual” seguem o mesmo princípio: Elas são contidas na memória física e apontadas por algum registrador de controle especial.

A descrição de regiões da memória não está restrita a apenas códigos e dados. Existem descritores especiais, chamados **descritores de sistema**. Entre eles, *call gates*, *task gates*, *task segment status* etc.

## Seletores e descritores de segmentos no modo x86-64

O modo x86-64 é um modo meio esquisito em alguns aspectos. Até então, tanto no modo real quanto no modo i386, os seletores fazem exatamente o que eles foram feitos para fazer: Selecionam um segmento de memória... No modo x86-64 eles quase não servem para nada!

Os registradores CS, DS, ES, FS, GS e SS continuam existindo, mas o único que realmente tem alguma utilidade é o CS porque guarda, em seu interior, o CPL e um bit dizendo em que modalidade do modo x86-64 o código será executado... Existem dois: O *long mode* e o *compatibility mode*. No modo *compatível* o processo comporta-se do mesmo jeito que o modo i386, exceto pelo fato de que temos acesso aos registradores extendidos de 64 bits.

Os demais seletores estão desabilitados no modo x86-64 puro. Se eles estiverem todos zerados (e, portanto, “inválidos”) ou com qualquer outro valor serão simplesmente ignorados:

```
/* readsel.c */
#include <stdio.h>

#define GET_SELECTOR(s) \
    __asm__ __volatile__ ( "movw %%" #s ",%ax" : "=a" ((s)) )

unsigned short cs, ds, es, fs, gs, ss;
```

<sup>15</sup> Existe uma outra tabela de descritores chamada *Local Descriptor Table*. Por motivos de simplicidade vou ignorar a existência dessa tabela na explicação que segue...

<sup>16</sup> Essa é uma exigência da arquitetura Intel para o modo protegido e não tem NADA haver com ponteiros NULL.

```

void main(void)
{
    GET_SELECTOR(cs);
    GET_SELECTOR(ds);
    GET_SELECTOR(es);
    GET_SELECTOR(fs);
    GET_SELECTOR(gs);
    GET_SELECTOR(ss);

    printf("CS=0x%04X, DS=0x%04X, ES=0x%04X, FS=0x%04X, GS=0x%04X, SS=0x%04X\n",
           cs, ds, es, fs, gs, ss);
}

-----%<---- corte aqui -----%<-----
$ gcc -o readsel readsel.c
$ ./readsel
CS=0x0033, DS=0x0000, ES=0x0000, FS=0x0000, GS=0x0000, SS=0x002B

```

No exemplo acima, note que DS, ES, FS e GS estão zerados. Mesmo assim o programa foi capaz de ler/escrever na memória, através do seletor DS! O registrador CS contém um valor e, de acordo com o diagrama da estrutura de um seletor que mostrei antes, ele aponta para o índice 6 da tabela de *descritores globais* (TI=0) e requisita o privilégio no *userspace* (RPL=3). O seletor SS parece desafiar o que foi dito antes. Ele contém um valor válido (índice 12 e RPL=3), mas isso está apenas para controle interno do sistema operacional e não tem o mínimo significado para os nossos programas. Quer dizer: não é usado pelo processador.

Se você executar o programinha acima no modo i386 obterá valores válidos para todos os seletores e alterá-los poderá levar ao erro de *segmentation fault*. No modo x86-64 puro, podemos alterar, sem medo, DS e ES. Os registradores FS e GS são especiais no sentido de que podem usar um descriptor como offset para um endereço linear, dependendo da configuração do processador. Isso não é usado no *userspace* e não nos interessa, no momento.

Neste ponto você pode estar se perguntando onde diabos foi parar a proteção, ou seja, os privilégios, no modo x86-64? Se os seletores não tem uso nesse modo, incluindo o RPL, como o processador sabe quais regiões pode ou não acessar se não há comparação com o DPL? Neste modo, o CPL continua sendo mantido internamente em CS, mas os privilégios das regiões de memória são mantidas nos *descritores de páginas*! O modo de paginação é **obrigatório** no modo x86-64 por esse motivo...

## Descritores no modo x86-64

Dos descritores usados por seletores, apenas os descritores de segmentos de código são necessários no modo x86-64 e apenas o seletor CS é considerado, os demais estão desabilitados.

As exceções são os seletores FS e GS que podem ser usados para apontar para endereços base nos descritores para os quais apontam. O seletor GS é particularmente interessante porque, no *kernelspace*, podemos usar a instrução SWAPGS que troca o endereço base pelo conteúdo da MSR<sup>17</sup> IA32\_KERNEL\_GS\_BASE (0xC0000102). Isso permite manter o endereço de estruturas acessíveis apenas pelo sistema operacional.

Ambos seletores, FS e GS, raramente aparecerão em seus códigos no *userspace*. A possível exceção é para o uso de FS no Windows: Na versão Win32 o seletor FS é usado pelo mecanismo chamado SEH (*Structured Exception Handler*), em blocos *try...catch*, em C++, quando o programa deixa o sistema operacional tratar algumas exceções (divisão por zero, por exemplo). Mas, suspeito que no modo x86-64 isso não seja feito desse jeito. De qualquer maneira, eu mesmo raramente vi FS e GS usados no *userspace*. De fato, raramente vi qualquer um dos seletores usados *explicitamente*.

---

<sup>17</sup> MSRs são registradores especiais mantidos pelo processador. A sigla vêm de *Machine Specific Register*.

## Cache de descritores

Sempre que um seletor é carregado um descritor é copiado para uma parte invisível e inacessível do seletor. Além dos 16 bits “visíveis” de um seletor, temos a estrutura de um descritor atrelada a ele, ou seja, seu limite, endereço base e bits de controle, DPL... Assim, sempre que usarmos um seletor o processador não tem que recarregar o descritor novamente.

Do ponto de vista do desenvolvedor isso é supérfluo, já que o processador fará esse *caching* de qualquer forma e essas informações adicionais são invisíveis.

## Proteção no modo x86-64

Desconsiderando a paginação, no modo x86-64 o processador assume que o endereço base sempre é zero e o tamanho do segmento corresponde a todo o espaço endereçável possível. Em teoria podemos acessar até 16 EiB (exabytes), ou  $2^{64}$  bytes, de memória. Na prática, as arquiteturas atuais permitem acesso de até 4 PiB (petabytes), ou  $2^{52}$  bytes<sup>18</sup>.

Mesmo que os campos de endereço base e o tamanho não sejam usados nos descritores, os privilégios e outros flags ainda são usados para o seletor CS.

Os seletores FS e GS, além do uso citado anteriormente, podem ser usados também para armazenarem um valor de 64 bits a ser usado como offset numa instrução. Por exemplo, se tivermos, no *userspace*:

```
lea rdi,[fs:rbx+rax*2]
```

Neste caso, se FS aponta para um descritor de onde a instrução obterá o endereço base e a somará ao conteúdo de RBX e ao dobro de RAX, colocando o valor calculado em RDI.

Se tentarmos usar prefixos de seletores, sem que sejam FS ou GS, em referências à memória, eles são sumariamente ignorados ou, no pior caso, causarão uma exceção de *instrução indefinida* (*undefined instruction*), já que alguns prefixos não estão definidos para o modo x86-64:

```
mov rax,[cs:rbx] ; Isso não está disponível no assembly x86-64 e,  
; causa uma exceção!
```

Já que seletores não são usados, as instruções LDS, LES, LSS, LFS e LGS **não existem** no modo x86-64... É claro que ainda podemos carregar os seletores via instrução MOV.

---

<sup>18</sup> Os bits excedentes têm que ser, necessariamente, uma cópia do bit 51, ou seja, um endereço no modo x86-64 tem sinal!



# Capítulo 2: Interrompemos nossa programação...

Citei exceções e faltas no capítulo anterior ao falar de *segmentation fault*, por exemplo. O mecanismo que o processador usa para o tratamento de erros é, em essência, o mesmo que ele usa para o tratamento de *interrupções*. Mas, o que é uma interrupção? E qual é a diferença entre uma interrupção e uma exceção ou falta?

## O que é uma interrupção?

O processador está constantemente executando código. Tanto o código do programa executado no *userspace* quanto o código contido no kernel. Não existe tal coisa como “processamento parado” ou “tempo ocioso” nos processadores<sup>19</sup>... Isso é uma abstração usada pelo sistema operacional ou ambientes gráficos. Em resumo: O processador está executando código o tempo todo, sem parar...

Durante a execução de um programa certos circuitos (por exemplo, o do teclado) podem pedir à CPU que o processamento normal seja **interrompido** para executar uma **rotina de tratamento de serviço** (ISR, *Interrupt Service Routine*). Essa rotina vai lidar com as necessidades do dispositivo que requisitou a atenção e, ao terminar, fazer com que o fluxo de processamento retorne ao normal. O teclado faz isso mudando o estado de um sinal elétrico, no processador, do nível baixo para o nível alto (as interrupções, no PC, são requisitadas na subida no “pulso”). Esse sinal é a chamada IRQ (*Interrupt ReQuests*).

Para suportar diversas IRQs diferentes, o seu computador tem um chip que interpreta os pedidos de interrupção dos dispositivos e os entrega à CPU juntamente com o número da requisição. Existem, atualmente, 15 IRQs diferentes e elas são numeradas de IRQ0 até IRQ15, onde IRQ2 não é usada. O número da requisição também está relacionado com a resolução de prioridade (IRQ1 tem mais prioridade do que IRQ7, por exemplo. No exemplo do circuito do teclado, ele está conectado à IRQ1).

Quando você digita uma tecla, o controlador do teclado (KBC – *KeyBoard Controller*) faz um pedido de interrupção IRQ1 ao *controlador programável de interrupções* (PIC). O PIC é programado de forma tal que, quando o processador reconhece o pedido (via pino INT# do processador), ele recebe um número correspondente à entrada da tabela de interrupções que deve ser usada para chamar a rotina de tratamento de serviço... Daí, o processador pára tudo o que está fazendo, salta para a ISR correspondente e, no final dessa rotina, indica ao PIC que tratou a interrupção escrevendo um comando EOI (*End Of Interruption*) num registrador especial do controlador. Logo em seguida, executa uma instrução IRET que fará o processador retomar a execução normal.

Assim como os descritores de segmentos, existe uma tabela que descreve cada uma das 256 entradas possíveis de interrupções. Essa tabela é chamada de IDT (*Interrupt Descriptors Table*).

No modo protegido as 32 entradas iniciais da IDT são reservadas para uso interno do processador. Essas entradas correspondem às exceções, faltas e abortos. As demais são livres para uso tanto por IRQs quanto por *interrupções por software* (via instrução INT).

## As Interrupções especiais: Exceções, Faltas e Abortos

19 Bem... não é bem assim! Mas, continue lendo, ok?

Algumas interrupções são causadas não por uma sinalização feita por um circuito externo, mas pelo próprio processador. É o caso de faltas como *General Protection Fault*, *Page Fault* ou *Stack Overflow Fault*. Se certas condições esperadas pelo processador forem violadas, ele interrompe o processamento normal e desvia o fluxo de processamento para tratadores especiais, que lidarão com essas “faltas”.

Existem três tipos especiais de interrupções de “falhas”: Exceções, Faltas e Abortos... A diferença entre os três termos é que um “aborto” é algo mais drástico que uma “falta”, que é mais drástico que uma “exceção”. Um erro de divisão por zero causará uma “exceção”. Um erro de validação de privilégios causará uma “falta”. Mas, existem erros que colocam o processador em um estado instável. Esses são os “abortos”, que geralmente não podem ser tratados (e levam à *Blue Screen Of Death* ou a um *Kernel Panic*).

Abortos só interessam ao sistema operacional. Estamos interessados apenas em algumas faltas e exceções.

### ***Existem dois tipos de interrupções de hardware diferentes***

Vimos, acima, o que é uma IRQ. Essas interrupções podem ser “mascaradas” (ou desabilitadas) facilmente. O flag IF no registrador RFLAGS toma conta disso, do lado do processador. Se o flag IF estiver zerado, o processador não aceitará nenhuma IRQ.

Para mascarar IRQs é só usar a instrução CLI, zerando o flag IF. Para habilitar as IRQs, basta executar a instrução STI, fazendo IF ser setado.

Existe também uma interrupção de hardware que não pode ser mascarada. Ela é chamada NMI (*Non Maskable Interrupt*) e está associada a um conjunto de erros que o seu computador pode enfrentar e são verificados por algum circuito externo... Geralmente não estamos muito interessados em NMIs.

**Atenção:** Quando zeramos o flag IF, o processador não aceitará as IRQs, mas isso não significa que o PIC não continue recebendo-as. Em certos casos o sistema operacional pode precisar “mascarar” interrupções no próprio PIC, além de zerar o flag IF. A mesma coisa acontece com a NMI... O processador é sempre obrigado a aceitar NMIs, mas podemos desabilitar o sinal elétrico em outro circuito do PC para inibi-lo (estranhamente, o chip controlador do teclado permite fazer isso!).

### ***Interrupções são sempre executadas no kernelspace!***

Não é possível criar rotinas de tratamento de interrupções no ring 3. Todas as interrupções são de responsabilidade do kernel. Assim, quando o processador recebe uma requisição de interrupção, o par de registradores SS:RSP , o registrador RFLAGS e também o par CS:RIP são empilhados, nesta ordem, e o controle é passado para o tratador de interrupção<sup>20</sup>. Quando a interrupção acabar, o kernel terá que devolver o controle ao seu código (que tem pilha própria e foi interrompido com os flags num estado conhecido).

Algumas exceções e faltas colocam na pilha, depois de empilhar CS:RIP, um código de erro. É o caso de *General Protection Fault* e *Page Fault*. O formato desse código de erro depende da falta (ou exceção). Uma GPF, por exemplo, recebe um erro diferente que uma *Page Fault*.

---

<sup>20</sup> SS também é empilhado por motivos de compatibilidade. Lembre-se ele não é usado no modo x86-64.

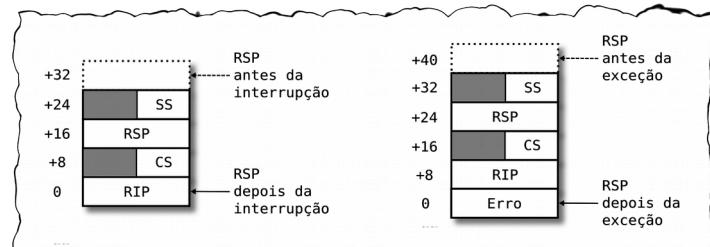


Figura 5: Pilha, antes e depois de uma interrupção ou exceção com código de erro.

Quando o tratador encontra uma instrução IRET ele recupera os conteúdos empilhados, na ordem em que os foram empilhados, saltando para o CS:RIP contido na pilha. Isso é mais ou menos como RET funciona, com alguns registradores a mais.

É responsabilidade das rotinas de tratamento de interrupções preservarem o conteúdo dos registradores de uso geral (exceto RSP) e recuperá-los antes de sair da interrupção.

### Faltas importantes para performance

A falta mais importante, do ponto de vista da performance, é a *Page Fault*. É bom lembrar que uma “página” é uma região de 4 KiB na memória mapeada numa tabela que permite o uso de “memória virtual”.

Essa falta acontece de acordo com uma das cinco condições abaixo:

- A página não está presente na memória física e, portanto, precisa ser mapeada;
- O nível de privilégio corrente (CPL) é menos privilegiado do que o descrito para a página;
- Tentar escrever em uma página *read-only*;
- RIP aponta para uma página marcada como “não executável”;
- Os bits reservados, no mapa de páginas, são diferentes de zero.

Conhecer essas regras é interessante, mas o mais importante é saber que, sempre que há um *Page Fault* o processador é interrompido e um tratador executado. Isso afetará outras regiões do processador como caches, por exemplo.

Dos motivos para a falta, o primeiro é o mais importante. É através dele que o processo de *page swapping* é feito. Assim, quanto mais faltas de página, provavelmente o sistema operacional estará mapeando páginas – possivelmente fazendo gravações e leituras em disco também. Não é de espantar que *page faults* possam ser o motivo de uma grande perda de performance. Temos que arrumar um jeito de evitá-las...

### Sinais: Interrupções no user space

Seu código em C pode implementar rotinas de tratamento de sinais. Esses sinais são “interrupções” no fluxo normal de operação do seu processo, isto é, seu programa pára o que está fazendo para atender um sinal e, quando a rotina de tratamento é finalizada, ele pode continuar o que estava fazendo (ou, em alguns casos, abortar o processo).

Sinais são usados em ambientes POSIX. O Windows, por exemplo, **não** implementa sinais<sup>21</sup>...

<sup>21</sup> Bem.... pelo menos a implementação não é tão boa. O MSDN indica que a função *signal()* é implementada para sinais como SIGINT, SIGABRT, SIGFPE, SIGILL, SIGSEGV e SIGTERM e é só. Mesmo assim, SIGINT não é suportado pelo Windows (embora *signal()* o permita). Outros sinais importantes como SIGKILL, SIGCHLD e

Um sinal é conhecido por um valor inteiro apelidado por *SIGxxx*. Abaixo temos uma lista com alguns dos sinais mais usados. Note que todo sinal, se não for tratado, tem um comportamento padrão:

Sinal	Descrição	Comportamento padrão
SIGINT	O usuário usou Ctrl+C no terminal.	Término do processo.
SIGKILL	Recebido quando o processo é “matado”.	Término do processo.
SIGSEGV	Uma referência inválida à memória foi feita.	Término do processo.
SIGTERM	Pedido de término do processo.	Término do processo.
SIGSTOP	Pedido de suspensão (parada) do processo (Ctrl+Z, no terminal?).	Suspensão do processo.
SIGUSR1, SIGUSR2	Sinais definidos pelo usuário.	Término do processo.
SIGCHLD	Processo filho (forked) foi terminado.	Ignorado.
SIGALRM	Sinal do timer. Útil para criar “timeouts” em processos. Esse sinal é programado via syscall <i>alarm()</i> .	Término do processo.

*Tabela 1: Alguns sinais mais conhecidos*

Os sinais SIGKILL e SIGSTOP não podem ter tratadores, ou seja, o comportamento padrão é fixo para esses sinais. SIGKILL é particularmente drástico. Ele mata o processo não importa o que esteja acontecendo. É comum, quando o usuário quer matar um processo de maneira definitiva, que use:

```
$ kill -9 8132
```

Neste exemplo, o comando *kill* envia um sinal SIGKILL (9) para o processo com PID 8132. O correto seria enviar o sinal SIGTERM e só se o processo não terminar, enviar SIGKILL:

```
$ kill -SIGTERM 8132
... espera um pouco ...
$ ps -eo pid | grep 8132 && kill -SIGKILL 8132
```

O motivo de SIGKILL não poder ser tratado é porque ele precisa mesmo ter o poder de terminar o processo. Os demais sinais podem ter tratadores que, se não chamarem a função *exit* antes de seu término, farão com que seu programa continue rodando de onde foi interrompido.

Suponha que você queira desabilitar o funcionamento do Ctrl+C. Basta fazer algo assim:

```
signal(SIGINT, SIG_IGN);
```

O símbolo SIG\_IGN é um tratador especial, pré concebido, que ignora o sinal. Existe também o símbolo SIG\_DFL, que indica o uso de um tratador *default* para o sinal. Nada impede que você crie sua própria rotina de tratamento. Elas têm sempre o seguinte protótipo:

```
void sighandler(int signal);
```

Onde, é claro, “sighandler” pode ser outro nome de função... Um exemplo, com relação ao SIGINT, se o usuário digitar Ctrl+C durante a execução de seu processo, poderia ser este:

---

SIGALRM são **inválidos** no Windows.

```

static void sigint_handler(int signal)
{
    printf("\nUsuário pediu interrupção!\n"
           "EU NÃO DEIXO!\n");
}

...
/* Em algum lugar do seu programa, registramos o manipulador para SIGINT: */
signal(SIGINT, sigint_handler);

```

É simples assim... Só preciso alertá-lo que a função *signal()* é obsoleta. O método preferido para registrar manipuladores de sinais é usando a função *sigaction()*. Esta função permite um ajuste fino do registro e manipulação de sinais. Ela permite registro de manipuladores que têm acesso a mais informações sobre o sinal e, inclusive, a possibilidade de bloquear outro sinal enquanto o tratador estiver em execução.

Em essência, *sinais* são interrupções...

### **Um exemplo de uso de sinais, no Linux**

Algumas funções não retornam até que alguma coisa aconteça. Essas funções são ditas “bloqueadas” pelo sistema operacional. Por exemplo, por default a função *recv()*, que lê um conjunto de caracteres vindos de um *socket*, não retorna até que tenha bytes lidos nos buffers do driver de rede... Uma maneira de criar uma rotina com suporte o recurso de *timeout*, usando *recv()*, é esta:

```

int alarm = 0;

/* Manipulador do sinal. */
int sigalarm(int signal) { alarm = 1; }

...
/* Assinala o handler de interrupção para SIGALRM. */
signal(SIGALRM, sigalarm);
...

/* Pede ao kernel para gerar um SIGALRM para o nosso processo em
   'timeout_in_seconds' segundos. */
alarm(timeout_in_seconds);

/* Neste ponto o seu processo pode ser "bloqueado". */
len = recv(fd, buffer, sizeof(buffer), 0);

/* Pede ao kernel para ignorar o pedido anterior, se a função acima
   não for bloqueada. */
alarm(0);

/* Se o alarme foi dado, faz algo! */
if (alarm)
{
    ...
    /* trata erro de timeout aqui */
    ...
}

```

Ao assinalar o manipulador *sigalarm()* ao sinal SIGALRM, quando este ocorrer, o processo não será terminado (comportamento default do SIGALRM), mas o processo “bloqueado” vai terminar e retornar... Logo depois de *recv()* colocamos um *alarm(0)* para dizer ao kernel que, se o SIGALRM não foi enviado, não o envie mais!

Se o sinal foi enviado, setamos a variável *alarm* para 1, indicando que o tempo limite já passou.

### **Não use *signal()*, use *sigaction()***

A função *signal()* é obsoleta e não deve mais ser usada. Ela ainda existe por motivos de

compatibilidade. O correto, hoje em dia, é usar a função *sigaction()*, que é bem mais flexível. Ela permite ajustar alguns *flags* que informam, por exemplo, se a função “bloqueada” que foi interrompida deve ser reiniciada ou não... Ainda, podemos ter tratadores de sinais que recebem mais informações, tornando-os ainda mais flexíveis. Ao invés de um tratador (*handler*), podemos ter uma ação (*sigaction*), onde a função receberá o número do sinal, um ponteiro para uma estrutura do tipo *siginfo\_t*, contendo um monte de informações e um terceiro argumento (geralmente não usado).

A função *signal()*, na realidade, usa *sigaction()* para fazer a sua mágica, mas o tipo de interrupção (se permite restart ou não da função “bloqueadora”, por exemplo), depende do sinal em si, não de algum ajuste cuidadoso que você possa fazer... Assim, use *sigaction()* para um melhor *fine tuning* só comportamento do tratador do sinal.

### ***Existe mais sobre sinais do que diz sua vã filosofia...***

Isso ai em cima é apenas um aperitivo sobre sinais. Sinais podem ser bloqueados, mascarados. Podemos tratar sinais em threads, etc... Consulte um bom livro sobre desenvolvimento para Unix como o material de *Richard W. Stevens* (“Advanced Programming in the UNIX Environment, Third Edition”, por exemplo).

# Capítulo 3: Resolvendo dúvidas frequentes sobre a Linguagem C

Antes de “cair de boca” no assembly é conveniente tentar acabar, de vez, com algumas dúvidas que estudantes de linguagem C têm. As principais, pelo que posso perceber, são sobre parametrização de funções e o uso de ponteiros. Aqui vai uma discussão sobre esses itens, que fazem o programador novato pensar que C é complicada e cheia de armadilhas.

## Headers e Módulos: Uma questão de organização

Alguns de meus leitores têm a dúvida recorrente sobre o porque da existência de arquivos com extensão “.h”... Num código fonte em C é comum termos diversos arquivos com extensão “.c” (chamados de “módulos”) e arquivos com extensão “.h” (chamados de “headers” ou “cabeçalhos”). Os módulos serão compilados, quase sempre separadamente, e depois linkados para montarem o executável ou biblioteca final. Os arquivos *header* existem para organizar melhor os códigos fonte. Eles **não são** bibliotecas de funções.

Repto e enfatizo: **Arquivos header não são bibliotecas!**

Bibliotecas existem em dois sabores: Estáticas e dinâmicas. Bibliotecas estáticas são conjuntos de funções que serão linkadas ao seu código fonte formando um programa monolítico, ou seja, todas as funções da biblioteca estática são incorporados na imagem binária contida no arquivo executável. Bibliotecas dinâmicas também são linkadas, mas o seu programa as carrega do disco, quando precisa delas.

No caso do Linux, as bibliotecas estáticas estão arquivadas num arquivo com extensão “.a”<sup>22</sup>. Já as bibliotecas dinâmicas ficam em arquivos com extensão “.so” chamados de “shared object”. Você pode listar esses arquivos em diretórios como */lib* e */usr/lib*.

No caso do Windows, bibliotecas estáticas têm extensão “.lib” e as dinâmicas são as velhas conhecidas DLLs. É comum que DLLs estejam armazenadas em arquivos com extensão “.dll”, mas alguns arquivos com extensão “.exe” também são DLLs disfarçadas (como o kernel, a GDI e a API para o usuário). Windows ainda disfarça DLLs em outras extensões (“.ocx”, por exemplo).

Se um arquivo *header* não é uma biblioteca, o que ele é? *Esses* arquivos geralmente contém declarações, protótipos de funções, macros, constantes definidas para o preprocessador, tipos, *externs* etc. Mas, não contém as definições de funções. Isso fica nas bibliotecas, externas ao seu programa, ou nos módulos, em arquivos “.c”. E a estrutura dos headers costuma ser a seguinte:

```
/* header.h */
#ifndef __HEADER_INCLUDED__
#define __HEADER_INCLUDED__

...
/* declarações, definições, macros, protótipos, etc */
...

#endif
```

O motivo da definição do símbolo `__HEADER_INCLUDED__`, no exemplo acima, deve-se a possibilidade de que o programador inclua o header diversas vezes ou, pior, o inclua de maneira circular. Por exemplo:

---

<sup>22</sup> A extensão “.a” vem de *archive*. O termo “arquivado” não foi escolhido levianamente na sentença.

```

/* header1.h */
#include "header2.h"
-----%<---- corte aqui -----%<-----
/* header2.h */
#include "header1.h"

```

Ao realizar as declarações dos *headers* desse jeito, somente se os símbolos estiverem definidos, garantimos que essas declarações serão feitas apenas uma única vez, não importa quantos #include você use.

Costumo nomear esses símbolos como *filename INCLUDED*, onde “filename” só possui o nome do arquivo, sem o “.h”. Como esses símbolos são “anônimos” (não possuem valores) e valem apenas para o preprocessador, eles não serão exportados para o executável final e, ao usar o “filename”, sei que ele será definido apenas para header específico. Claro que você pode usar o padrão que seja mais conveniente para seu projeto. De fato, algumas IDEs usam alguns padrões “malucos”. O importante é a característica única na nomeação dos símbolos.

Nada te impede de definir funções dentro de um *header*, mas essa não é uma boa prática... Existem exceções à regra: Algumas “funções” intrínsecas nos headers do GCC definem funções inline dentro de headers (veja “cpuid.h”, por exemplo). Eu recomendo que você não faça isso em seus códigos. Acompanhar onde algumas declarações são feitas, em projetos grandes, já é complicado o suficiente sem que se quebre uma regra tão básica.

## Chamadas de funções contidas no mesmo módulo

Num código como abaixo você espera que a função *f* seja chamada pela função *g*, mas não é isso que geralmente acontece:

```

/* teste.c */

int f(int x) { return x + x; }
int g(int x) { return x * f(x); }

```

Em casos como esse o compilador tende a criar uma função *f*, que pode ser chamada por funções contidas em outros módulos, e **incorporá-la**, ou seja, codificá-la *inline* na função *g*.

Com isso você acaba com duas cópias do mesmo código: Uma que pode ser chamada e outra que foi incorporada. Veja como fica, em assembly:

```

f:
    lea    eax,[rdi+rdi]
    ret

g:
    lea    eax,[rdi+rdi]      ; Deveria ser "call f".
    imul   eax,edi
    ret

```

Aqui não há grandes problemas, mas imagine que *f* seja uma rotina grande com umas 200 instruções. Ao invés de um simples CALL em *g*, teríamos duas cópias de *f*. Isso, algumas vezes, é inaceitável.

Para evitar esse comportamento você pode fazer duas coisas: Mudar o atributo da função *f* ou codificar essa função em um outro módulo. Funções definidas em módulos diferentes são, necessariamente, chamadas via CALL, a não ser que sejam marcadas como *inline* (mas, fique ciente, essa “marca” é apenas uma dica!).

Para mudar um atributo de uma função, no GCC, basta atrelar a declaração *\_attribute\_((attr))* no início da declaração, onde *attr* é um dos atributos que o GCC aceita. No caso, estamos interessados no atributo *noinline*:

```
__attribute__((noinline)) int f(int x) { return x+x; }
int g(int x) { return x*f(x); }
```

## Tamanho de inteiros e ponteiros

Se você já usou C com Windows, deve ter se perguntado: Pra que diabos existe o tipo 'long'? Afinal, no Windows, 'long' e 'int' têm exatamente o mesmo tamanho e a mesma semântica (ambos são inteiros de 32 bits de tamanho). Isso também é válido para o modo 32 bits da maioria dos sistemas operacionais. No modo x86-64 a coisa é mais complicada...

Existem quatro modelos de uso dos tipos inteiros para o modo x86-64, dependendo do sistema operacional em uso. Elas são conhecidas por siglas: IL32P64 (ou LLP64), LP64 (ou I32LP64), ILP64 e SILP64. Nessas siglas o 'I' corresponde ao tipo 'int', 'L' ao 'long' e 'P' à 'Ponteiro'. I32LP64 significa “int de 32 bits; long e ponteiros de 64 bits”, por exemplo.

Eis uma tabela mostrando as diferenças, em bits, entre os tipos dos quatro modelos e quem os usa (excluí os ponteiros já que nos quatro modelos eles têm sempre 64 bits de tamanho):

Modelo	short	int	long	Sistema Operacional
IL32P64	16	32	32	Windows
I32LP64	16	32	64	POSIX ABI
ILP64	16	64	64	HAL (“Hello, Ave!”)
SILP64	64	64	64	UNICOS

Tabela 2: Modelos de inteiros e sistemas operacionais

A maioria dos sistemas operacionais de 64 bits atuais, que são baseados em POSIX, diferenciam os tipos 'long' e 'int'. Mesmo assim, o tipo 'long long' foi incorporado na especificação de C para garantir que tenhamos um tipo explícito relacionado com 64 bits. O modelo I32LP64 torna esse novo tipo “obsoleto”. Ainda, a especificação da linguagem C preocupou-se com essas diferenças. Para isso existe o header *stdint.h* que define apelidos para tipos como: *int8\_t*, *int16\_t*, *int32\_t* e *int64\_t* e seus derivados *unsigned*, colocando um 'u' na frente do nome do tipo (exemplo: *uint64\_t*). Se você quer criar códigos que cruzem plataformas é recomendável que use esses tipos.

Os modelos ILP64 e SILP64 são usados por sistemas “obscuros” (HAL Computer Systems [Ave! O que você está fazendo, Ave?!] e UNICOS, segundo o wikipedia). Não é estranho que, no caso do modelo SILP64, 'short' e 'long' tenham o mesmo tamanho?

## Não é prudente confiar nos tamanhos dos tipos default

Talvez você tenha se acostumado com as novas arquiteturas de processadores onde os tipos *char*, *short*, *int*, *long* e *long long* tenham tamanhos definidos. Não é uma boa idéia confiar nisso se você pretende criar código para várias plataformas.

Em primeiro lugar, **não existem** tipos *short*, *long* e *long long*. Essas são variações de tamanho do tipo *int*. Em segundo lugar, esses tamanhos variam de acordo com o processador. Eis um exemplo na tabela à seguir:

<b>Tipo</b>	<b>Z-80</b>	<b>8086</b>	<b>386 e superiores</b>
<i>char</i>	8	8	8
<i>short int</i>	8	8?	16
<i>int</i>	8	16?	32
<i>long int</i>	16	16	$32^{23}$

*Tabela 3: Tamanho de tipos por processador*

Num processador de 8 bits como o Z-80 quase todos os tipos de inteiros têm 8 bits de tamanho. Faz sentido, porque trata-se de um processador de 8 bits! Já no 8086, provavelmente o tipo *int* tem 16 bits e essa será a diferença entre *short* e *long*... Já no 386 o tipo *short int* é diferente do *char*.

Marquei com “?” Os tamanhos, em bits, que não tenho mais certeza (faz tempo que não lido com esses processadores).

É essencial que você consulte a documentação do compilador para o processador alvo ou faça uso do header *limits.h* que contém constantes como CHAR\_MAX, SHORT\_MAX, INT\_MAX e LONG\_MAX, explicitando o máximo valor que pode ser armazenado nesses tipos. Aliás, sequer podemos supor que um *char* tenha sempre 8 bits... Em computadores antigos (alguns *mainframes*, por exemplo, um *char* pode muito bem ter 7 bits). Para isso *limits.h* fornece a constante CHAR\_BITS.

Outro detalhe é o uso do operador *sizeof*. Ele sempre lhe dará o tamanho em bytes de um tipo, até mesmo ponteiros. Sendo um operador, às vezes ele não pode ser usado no pré-processador do compilador. Fazer algo como mostrado abaixo causará erro de pré-compilação:

```
#if sizeof(char *) != 8
...
#endif
```

Daí a importância dos símbolos definidos em *limits.h*.

### **Existe diferença entre “declarar” e “definir” um símbolo**

Primeiro, vamos a uma definição de “símbolo”: Todos os nomes que você dá as variáveis, funções e tipos são genericamente conhecidos como “símbolos”. Um símbolo é um nome ou “apelido”.

A principal coisa que você deveria entender sobre a linguagem C (e C++) é que existe uma diferença essencial entre declarar algo e definir algo. Quando você faz:

```
unsigned long x;
```

Você está **declarando** um símbolo chamado “x” que será usado para conter um valor inteiro, longo e sem sinal. Repare que você não está dizendo para o compilador qual é esse valor, apenas está “declarando”: *Ei, compilador! Separa um espaço para um inteiro longo sem sinal e o chame de 'x'!*

Quando atribui um valor, você **define** o conteúdo da variável. Simples assim...

No caso de variáveis “comuns”, como mostrado acima, o símbolo ‘x’ é automaticamente **definido** como contendo um valor aleatório, imprevisível.

Existe também um atalho onde você pode declarar e definir um símbolo ao mesmo tempo:

```
int x = 2;
```

Isso é um atalho! O que você está realmente fazendo é declarar o símbolo “x” como sendo do tipo

---

23 Como já sabemos, no modelo I32LP64, no modo x86\_64, este tamanho pode ser de 64 bits!

“int” e, logo em seguida, definindo o seu conteúdo como tendo o valor 2.

Então não parece haver muita diferença entre *declarar* e *definir*, certo? A diferença está nos casos onde não é possível definir um símbolo porque já foi definido em outro lugar. É ai que entram os arquivos *header*. Nesses arquivos é comum existirem **declarações** de funções e variáveis que serão definidas em algum outro lugar... A **declaração** abaixo:

```
int printf(const char *fmt, ...);
```

Informa ao compilador que, em algum lugar, existe uma função chamada *printf* que toma um ponteiro para um *char*, constante, e uma lista variável de parâmetros, retornando um *int*. No contexto do uso dessa função não é necessário saber onde ela foi **definida**. Esse é um tipo de declaração que você encontra em *headers* como *stdio.h*.

## Diferença entre declaração e uso

Quando você usa operadores de ponteiros, existe uma diferença entre declará-lo e usá-lo. As duas coisas, abaixo, são diferentes:

```
int *p = &x; /* declarando um ponteiro que conterá o endereço da variável 'x'. */
*p = 10;      /* Usando o operador de indireção para colocar 10 dentro do
                 endereço contido na variável 'p' (declarada, acima, como ponteiro!). */
```

A primeira linha declara o ponteiro ‘p’ (usando o ‘\*’) e inicializa a variável ‘p’ com o endereço de ‘x’. A mesma coisa pode ser feita assim:

```
int *p;
p = &x;
```

O ponto é que, na declaração, o ‘\*’ não é um operador, mas um artifício sintático, usado para declarar ponteiros. Já no uso da variável “p”, o asterisco é um **operador de indireção**. É uma operação que mandamos o compilador fazer com a variável “p”... Falo mais sobre ponteiros adiante...

## Parênteses

Pode parecer ridículo falar de parênteses, mas eles têm vários significados num código fonte em C. Eles podem ser usados para especificar parâmetros de funções, podem ser usados para forçar a ordem de cálculo numa expressão, podem ser usados para adequar um “tipo” de variável a outro, podem ser usados para realizar uma chamada a uma função e podem ser usados para resolver expressões que são ambíguas:

```
int f(int);      /* Declaração de protótipo. Função aceita um parâmetro do tipo 'int'. */
int (*p)[10];    /* Declara ponteiro de array para 10 'int's.
                   Sem os parênteses teríamos um array de ponteiros. */
z = (x + 3) * y; /* Sem os parênteses o 3 seria multiplicado ao 'y'! */
i = (int)c;       /* Converte a variável 'c' para o tipo 'int'. */
f(2);            /* Chama a função f passando o valor 2 como parâmetro. */
```

Nos dois primeiros casos temos declarações onde os parênteses são necessários. Nos dois últimos, temos **operadores** diferentes (casting e chamada de função) e no terceiro caso temos uma resolução de ambiguidade.

## Escopo dos parâmetros de funções

Uma função em C pode receber parâmetros vindos de outras funções que a chamam. Uma função sempre é chamada por outra função. Assim, temos a função **chamadora** e a função **chamada**. Não

há mistério nisso.

Toda função pode receber parâmetros. E há um detalhe importante: Parâmetros de funções são encarados como se fossem variáveis locais à própria função e só podem ser alterados dentro dela. Veja um exemplo de uma simples função que preenche um buffer com zeros:

```
/* Primeira versão da função.  
   Usa os parâmetros como se fossem inalteráveis. */  
void fillzeros1(char *ptr, size_t size)  
{  
    size_t i;  
  
    for (i = 0; i < size; i++)  
        ptr[i] = 0;  
}  
-----%<---- corte aqui -----%<----  
/* Segunda versão.  
   Usa os parâmetros como variáveis locais. */  
void fillzeros2(char *ptr, size_t size)  
{  
    while (size--)  
        ptr[size] = 0;  
}
```

É fácil provar que os parâmetros são locais à função:

```
size_t s;  
char buffer[1024];  
  
s = sizeof(buffer);  
fillzeros2(buffer, s);  
  
printf("O valor de 'size' é %d\n", s);
```

No fim das contas o fragmento de rotina, acima, imprimirá o valor 1024, mesmo que 'size' seja alterado de dentro da função *fillzeros2*. Isso é condizente com o fato de que poderíamos chamar a função passando um valor constante, ao invés de uma variável:

```
fillzeros(buffer, 1024);
```

O segundo parâmetro, neste caso, é uma constante que, por definição, não pode ser alterada. Mas, dentro da função esse valor é colocado na variável local 'size'.

Se essa explicação não bastar, historicamente parâmetros são passados pela pilha, ou seja, uma cópia dos parâmetros são empilhados e usados pela função. As variáveis e constantes originais, passados à função não são tocados por ela.

## Protótipos de funções

É comum, num código mais complexo escrito em C ou C++, que você veja declarações de funções de maneira “incompleta”. Isso serve para avisar o compilador o que está por vir sem que você tenha que definir uma função com antecedência. Provavelmente isso não é nenhuma novidade para você. O detalhe é que, às vezes, essas declarações incompletas podem parecer meio confusas:

```
extern void qsort(void *, size_t, size_t, int (*)(const void *, const void *));
```

Onde estão os nomes dos parâmetros na declaração acima? Onde está o corpo da função? E esse quarto parâmetro maluco? No caso específico do protótipo do *qsort* estamos dizendo ao compilador que a função tomará quatro parâmetros. O primeiro é um ponteiro genérico (*void \**), os outros dois são do tipo *size\_t* e o quarto é um ponteiro para uma função que toma dois parâmetros “*const void \**” e devolverá um inteiro. Essa declaração também diz que *qsort* não retorna valores (*void*) e é definida em algum outro lugar, talvez numa biblioteca.

O que essa declaração não diz é qual serão os nomes das variáveis locais associadas aos parâmetros! É isso! Não há nenhuma informação adicional para a função nessa declaração. Os protótipos só existem para dizer ao compilador: “Eu ainda não sei quem é esse cara, mas ele se parece com isso ai!”. Fornecer o nome dos parâmetros é opcional.

Protótipos, em minha opinião, são construções excelentes para organizar código. Costumo usá-los para colocar minhas funções em uma ordem lógica em meus códigos-fonte. Por exemplo:

```
#include <stdio.h>
#include <stdlib.h>

/* Nesse meu código eu uso uma função calc(), mas ela
   será definida somente DEPOIS da função main(). */
int calc(int);

int main(int argc, char *argv[])
{
    int valor;

    if (argc != 2)
    {
        fprintf(stderr, "Uso: test <valor>\n");
        return 1;
    }

    valor = atoi(argv[1]);

    /* A função main() precisa saber como é a função calc() para
       poder usá-la! Dai o protótipo, lá em cima. */
    printf("O triplo do valor %d é %d\n", valor, calc(valor));

    return 0;
}

/* Finalmente, calc() é definida aqui. */
int calc(int valor) { return (valor + valor); }
```

É claro que não faz nenhum mal colocar o nome dos parâmetros no protótipo. Aliás, é até uma boa ideia, já que o compilador fará testes de sintaxe entre a declaração (do protótipo) e a definição da função. Mas, ao mesmo tempo, ao não colocar nomes de parâmetros, você se libera de ter que certificar-se que todos os arquivos que contenham as definições de suas funções tenham que ter, em algum outro lugar, protótipos com parâmetros nomeados exatamente como na definição. Eu prefiro não nomear parâmetros nos protótipos. Digamos que essa é uma prática tradicional entre os programadores que lidam com a linguagem C...

## **Problemas com algumas estruturas com máscaras de bits**

A especificação da linguagem C nos diz que o uso de máscaras de bits é algo “dependente de implementação”. Ao fazer algo assim:

```
struct mystruc_s {
    unsigned x:6;
    unsigned y:27;
};
```

Você pode esperar que o primeiro bit do membro 'y' seja colocado imediatamente depois do último bit do membro 'x'. Em algumas arquiteturas isso pode ser verdade, mas não quando falamos das arquiteturas Intel e do compilador GCC (e alguns outros). A estrutura acima produzirá dois DWORDs, onde os 6 bits inferiores estarão no primeiro DWORD, atribuído ao membro 'x', e os 27 bits seguintes no segundo DWORD, atribuído ao membro 'y'.

Para evitar essa divisão, podemos pedir ao compilador para “compactar” os bits:

```
struct mystruc_s {
    unsigned x:6;
    unsigned y:27;
} __attribute__((packed));
```

Agora a estrutura terá exatamente 5 bytes. Os 33 bits estarão compactados, o mais próximo possível, colocando os primeiros 32 bits dentro de um DWORD e o bit que sobra no último BYTE.

Na maioria das vezes usar máscaras de bits não é uma grande ideia. O compilador vai gerar muito código para arranjar os valores desejados. Imagine que você use a estrutura compactada, acima, e queira ler os valores de x e y:

```
struct mystruct_s s = { 2, 3 };
unsigned a, b;

...
a = s.x;
b = s.y;
-----%<---- corte aqui -----%<-----
; Código gerado pelo compilador...

; O arranjo binário dos valores x e y da estrutura (msb -> lsb):
;      ??????y yyyyyyy yyyyyyy yyyyyyy yxxxxxx
;      |                               |
; bit 31                           bit 0

; Lê o primeiro byte e isola x.
movzx eax,byte ptr [s]
and   eax,0x3F
mov   [a],eax                   ; coloca em 'a'.

movzx eax,byte ptr [s]
movzx ecx,byte ptr [s+1]

; coloca os 2 primeiros bits de y no lugar certo.
shr   al,6
movzx eax,al

; coloca os 8 bits do próximo byte de y no lugar certo.
sal   rcx,2
or    rcx,rax

; coloca os próximos 16 bits de y no lugar certo.
movzx eax,byte ptr [s+2]
sal   rax,10
or    rax,rcx
movzx ecx,byte ptr [s+3]
sal   rcx,18
or    rcx,rax

; e finalmente coloca o último bit de y no lugar certo.
movzx eax,byte ptr [s+4]
and   eax,1
sal   rax,26
or    rax,rcx

; armazena o valor de y.
mov   [b],eax
```

Que código horrível! Só como comparação, eis a implementação que **eu** faria:

```

; Lê 64 bits da memória...
movzx rax,byte ptr [s]
mov rcx,rax           ; Guarda valor lido em RCX.

; Isola os 6 bits inferiores.
and eax,0x3f
mov [a],eax

; Pega os 27 bits restantes.
shr rcx,6
and ecx,0x7fffffff
mov [b],ecx

```

Mas o ponto é que, ao usar mapas de bits, o seu código vai ficar enorme e, é claro, um tanto lento. Isso é especialmente válido com estruturas compactadas. Observe agora o código equivalente com a estrutura **sem** o atributo *packed* (gerado pelo compilador):

```

mov eax,[s]
and eax,0x3f
mov [a],eax

mov eax,[s+4]
and eax,0x7fffffff
mov [b],eax

```

É bem parecido com minha implementação, não é? O motivo de usar EAX ao invés de RAX, é que a estrutura não tem 64 bits de tamanho... Mas, na verdade, já que dados também são alinhados, não há problemas em ler um QWORD onde temos apenas 33 bits (um bit a mais que um DWORD). Por isso minha rotina pode ser um pouquinho mais rápida que a gerada pelo compilador...

Mesmo assim, sem a compactação da estrutura, o código do compilador fica, óbviamente, mais simples e rápido em relação ao código anterior.

Siga a dica: Evite campos de bits sempre que puder.

## **Finalmente: Ponteiros!**

Uma das coisas que causa certo desespero às pessoas que tiveram o primeiro contato com linguagens como C e C++ são as “criaturas do inferno” conhecidas como ponteiros. Espero poder infundir um pouco de autoconfiança e mostrar que ponteiros não mordem, são quase uns anjinhos, e que são um dos recursos mais úteis que outras linguagens fazem questão de esconder.

Sempre que você ler a palavra “ponteiro” pode, sem medo, substituí-la por “endereço de memória”, ou simplesmente “endereço”. E saiba que todo símbolo, exceto por macros e definições feitas no nível do preprocessador, é um ponteiro, mesmo que não pareça. O fragmento de código, abaixo, exemplifica:

```

int x;
...
x = 10;
...

```

Quando declaramos a variável global 'x' estamos pedindo ao compilador que crie um espaço na memória com o tamanho de um *int* (4 bytes) e “apelide” o endereço inicial desse espaço de 'x'. Isso quer dizer que 'x' só existe no seu código fonte e dentro do compilador. O nome real dessa variável é o seu endereço. Assim, quando atribuímos o valor 10 à variável 'x' o compilador entende que deverá gravar o valor 10 no endereço cujo apelido é 'x'. Ele faz, mais ou menos isso:

```
mov dword ptr [0x601024],10
```

Neste exemplo, o endereço 0x601024 é um exemplo de onde o compilador pode decidir colocar a

variável, na memória.

Mesmo em assembly, que é uma linguagem de nível maior que a “linguagem de máquina”, podemos usar a metáfora da variável. Sem nos preocuparmos muito com a sintaxe do código abaixo, podemos usar um símbolo como apelido para um endereço, como se fosse uma variável:

```
...
section .bss
x: resd 1      ; 'x' é um símbolo que equivale a um
; endereço de memória da sessão de dados onde
; reservamos o espaço para um dword.

section .text

...             ; Coloca o valor de 32 bits '10' no endereço
; apelidado por 'x'.
...
...
```

## Declarando e usando ponteiros

Em C o conceito de “ponteiro” é um pouco mais especializado. Trata-se de uma variável que contém um endereço, ao invés de um valor. Uma variável do tipo ponteiro faz referência a um dado contido na memória, de maneira indireta, isto é, essa variável “especial” (o ponteiro) contém um endereço que “aponta” para um dado do tipo declarado. Quando fazemos algo assim:

```
int * p;
```

Dizemos que a variável 'p' será usada para conter um endereço para alguma região da memória (ou seja, é um ponteiro!) que possui um dado do tamanho de um *int*. Mas, atenção: A declaração acima não inicializa a variável! Um exemplo melhor está no gráfico abaixo:

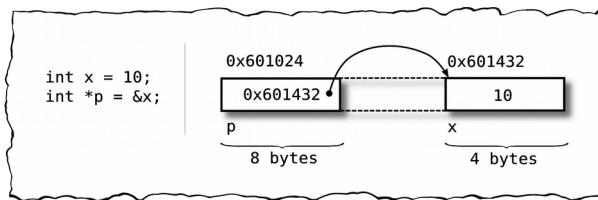


Figura 6: Ponteiro que aponta para uma variável

Aqui o compilador escolheu o endereço `0x601432` para a variável 'x' e o endereço `0x601024` para a variável 'p'. Só que essa última é um ponteiro que foi inicializada com o endereço de 'x' (via operador `&`).

Note... a variável é “p”, e não “\*p”!

Por questão de estilo, prefiro **declarar** ponteiros usando o '\*' próximo ao nome da variável. Isso serve apenas para “embelezar” o código. Se você preferir separar a declaração do jeito que fiz acima, por motivos de clareza, fique à vontade...

Conforme vimos, em relação aos modelos IL32P64 e I32LP64, na arquitetura x86-64, todo ponteiro, não importa o tipo associado a ele, tem exatamente 8 bytes de tamanho (o tamanho de um QWORD, ou seja 64 bits).

Eis um exemplo simples de declaração e uso de um ponteiro:

```

#include <stdio.h>
int x = 10;
int main(int argc, char *argv[])
{
    /* declaração de ponteiro 'p' definido como tendo o endereço de 'x'. */
    int *p = &x;
    printf("%lu\n%d\n", p, *p);
    return 0;
}

```

O código acima imprimirá um valor “aleatório” – um endereço escolhido pelo compilador – e o valor 10. Mas, o que significa esse `*p` usado como parâmetro na chamada de `printf`?

Diferente da declaração da variável, esse outro `*` é um **operador de derreferência ou *indireção***<sup>24</sup>. O conceito é que uma variável do tipo ponteiro contém uma referência (o endereço) usada para obter o dado apontado e, ao usar o operador `*`, você diz ao compilador: “pegue o dado cujo endereço está no ponteiro”.

Repare que existem dois momentos em que lidamos com ponteiros:

1. A declaração/definição: Onde reservamos o espaço para a variável;
2. O uso: Quando usamos o operador de indireção para obter o dado apontado.

A declaração de um ponteiro é feita como qualquer outra variável, exceto que o tipo é seguido de um `*` (ou, o nome da variável é precedido pelo asterisco, dependendo de como você encara). E, também, como para qualquer variável, tudo o que o compilador fará é alocar espaço suficiente para caber um endereço.

O tipo associado ao ponteiro é usado em operações aritméticas envolvendo o ponteiro. Ponteiros são valores inteiros e as mesmas regras da aritmética, válidas para *unsigned longs*, são válidas para eles: Podemos incrementar ou decrementar um ponteiro; podemos somar dois ponteiros (embora isso não seja lá muito útil!); podemos somar um valor inteiro a um ponteiro; podemos subtrair dois ponteiros (isso é útil!) ou uma constante...

No exemplo abaixo assumo que o ponteiro `'p'` foi inicializado em algum outro lugar. Esse fragmento serve para responder a pergunta: O que acontece quando incrementamos um ponteiro declarado como sendo do tipo `int`?

```

/* Declarei como 'extern' para dizer que esse ponteiro foi inicializado em
   algum outro lugar... */
extern int *p;
...
p++;

```

Vamos supor que o endereço contido em `'p'` seja 0x601040. Depois de incrementá-lo, **não** obteremos 0x601041, mas sim 0x601044! Isso acontece por causa do tipo associado ao ponteiro. Um `int` possui 4 bytes de tamanho (32 bits), então o ponteiro é incrementado de 4 em 4. Se tivéssemos ponteiros de tipos mais complexos, como no exemplo:

```

/* Essa estrutura tem 52 bytes de tamanho. */
struct vertice_s {
    float x, y, z, w;
    float nx, ny, nz;
    float r, g, b, a;
    float s, q;
};

```

---

<sup>24</sup> Chamarei de operador de *indireção* daqui pra frente, pois “dereferência” parece ser uma palavra inexistente na língua portuguesa.

```

struct vertice_s *vrtxp, *p;
...
p = vrtxp + 10;

```

Assumindo que 'vrtxp' tenha sido inicializado em algum outro lugar, o ponteiro 'p' conterá o endereço contido em 'vrtxp' mais 520 bytes, já que o tipo 'struct vertice\_s' tem 52 bytes de tamanho.

Então, recapitulando:

```

char *p;      /* isso é uma declaração de um ponteiro 'p' não inicializado. */
char *p = &s; /* Isso é a declaração de um ponteiro 'p' inicializado com o
               endereço de 's'. */
p = &s;        /* Isso coloca o 'endereço de' 's' no ponteiro 'p'. */
*p = 'a';     /* Isso coloca a constante 'a' no endereço contido no ponteiro 'p'. */
p++;          /* Isso incrementa o endereço contido em 'p'. */

```

## **Problemas com ponteiros para estruturas**

Há um problema com o uso do operador de indireção e a notação de estruturas (e uniões). O operador de membro de dados '.' tem maior precedência do que o operador de indireção '\*'. Por causa disso o código abaixo não faz o que você espera:

```

struct S {
    int x;
};

/* Apenas um exemplo de inicialização de ponteiro para estrutura. */
struct S *p = &s;
...

/* Isso não faz o que você pensa que faz! */
*p.x = 3;

```

Acabaremos com algum erro de compilação. É de se supor que "\*p.x" signifique a derreferência do ponteiro 'p' seguida da obtenção do membro de dados 'x', mas, como '.' tem precedência maior, o compilador entende que 'p.x' é o ponteiro. O que é falso!

Existem duas maneiras de resolver isso: Podemos usar parênteses para acabar com a ambiguidade:

```
(*p).x = 3; /* Essa é a sintaxe correta! */
```

Ou, desde o padrão ANSI da linguagem C existe um atalho. O operador '->' é usado para obter um membro de uma estrutura através de um ponteiro. Ou seja, o que está à esquerda de '->' deve ser sempre um ponteiro para uma estrutura e à direita, um membro de dados. A mesma linha, acima, fica assim:

```
p->x = 3;
```

A coisa funciona do mesmo jeito para membros de uniões.

## **Ponteiros e strings**

Você já deve ter topado com uma string, em C. São aquelas constantes cercadas por aspas duplas (""). Sinto dizer, já que a contradição na próxima sentença vai dar um "nó" no seu cérebro:

Uma string não é uma string! É um array de *chars* terminado com o byte 0 (zero)!

O que ocorre é que as funções da biblioteca padrão da linguagem C interpretam arrays de *chars* terminados com zero como se fossem strings. Isso torna a linguagem flexível, já que podemos usar qualquer array como se fosse uma string, basta fazer um *casting* de qualquer ponteiro para 'char \*', e nos certificarmos que, em algum ponto, o array contenha um byte zero.

Fica bastante óbvio que strings não existem, como um tipo primitivo, em C, já que não existem

operadores especializados para lidar com elas. Ou você as trata como arrays, diretamente, ou lida com elas através de funções da biblioteca padrão como *strcpy*, *strcat*, *strupr* etc.

## **Diferenças entre declarar ponteiros e arrays contendo “strings”**

Existem alguns jeitos de declararmos e inicializarmos “strings”:

```
char *str1 = "Hello";
char str2[] = "Hello";
char str3[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Sempre que topar com uma constante do tipo string, cercada por aspas duplas, o compilador traduzirá isso para um array contendo os caracteres individuais e um '\0' (zero) adicional. Assim, a primeira declaração é a de um ponteiro que conterá o endereço do array que contém 6 bytes ('H', 'e', 'l', 'l', 'o' e '\0').

As declarações de str2 e str3 são equivalentes entre si. Elas dizem para o compilador que os 6 bytes devem ser alocados na região de dados da memória e inicializados com 'H', 'e', 'l', 'l', 'o' e '\0'. Depois disso os símbolos str2 e str3 conterão os endereços iniciais dos arrays.

**Atenção:** A declaração abaixo não coloca o valor zero no final do array. A constante tem 6 bytes (contando com o zero final), mas o array foi explicitamente declarado como tendo 5 chars (bytes):

```
char str4[5] = "Hello";
```

O compilador colocará os 5 primeiros bytes da constante no array, descartará o sexto byte e, talvez, emita um aviso sobre isso.

Existe uma diferença entre declarar um ponteiro que aponta para uma constante e um array inicializado com uma. E é sutil: No primeiro caso, a constante não pode ser alterada porque, afinal de contas, é uma **constante!** No caso dos arrays a constante é usada para inicializar o conteúdo da memória reservada para eles e pode, em *runtime*, ser modificada.

Para ilustrar, eis um programa que falhará miseravelmente, explodindo o famigerado “Segmentation Fault” em nossas caras, provavelmente porque constantes, quando são armazenadas na memória, ficam em uma sessão *read-only* do código:

```
/* strtok.c */
#include <stdio.h>
#include <string.h>

/* Isto é um ponteiro para uma constante! */
char *s = "Hello, world!";

int main(int argc, char *argv[])
{
    char *p;

    p = strtok(s, ", ");
    /* Ocorrerá um 'segmentation fault' aqui! */
    while (p != NULL)
    {
        printf("%s\n", p);
        p = strtok(NULL, ", ");
    }

    return 0;
}
-----%----- corte aqui -----$-----
```

```
$ gcc -g -o strtok strtok.c
$ gdb ./strtok
(gdb) r
Starting program: strtok

Program received signal SIGSEGV, Segmentation fault.
strtok () at ../sysdeps/x86_64/strtok.S:190
190    .../sysdeps/x86_64/strtok.S: No such file or directory.
(gdb) bt
#0  strtok () at ../sysdeps/x86_64/strtok.S:190
#1  0x00000000040056f in main (argc=1, argv=0x7fffffff1c8) at strtok.c:11
```

O problema com esse programa é que *strtok* espera poder escrever no array apontado por 's' e não consegue. Se você mudar a declaração de 's' para:

```
char s[] = "Hello, world!";
```

Tudo funcionará perfeitamente.

## ***Mais ponteiros e arrays***

Assim como com ponteiros, existe uma diferença entre declaração e uso. No uso de arrays, o operador de indexação '[' – usado para obter um item de um array – é, na verdade, um atalho para aritmética de ponteiros.

Esse operador toma um ponteiro como base e um índice que é somado a esse ponteiro para obter um endereço. De posse desse endereço calculado, o dado é acessado. Os dois usos, abaixo são equivalentes:

```
int x[10];
...
y = x[3];      /* 'x' é um ponteiro e 3 é um deslocamento. */
y = *(x + 3); /* É a mesma coisa!!! */
```

O nome da variável do array 'x' é um ponteiro para o primeiro item do array. Podemos chamar esse símbolo de “base” do array. O índice, dentro de '[ ]', é o deslocamento adicionado a essa “base”, levando em conta o tipo do ponteiro, para obter o endereço do item desejado. Por isso, para inicializar um ponteiro apontando para o primeiro item de um array, podemos sempre fazer de uma das duas maneiras:

```
int x[10];
int *p;

p = x;          /* 'p' aponta para o início do array. */
p = &x[0]; /* É a mesma coisa!!! */
```

Ou seja, se você não se sente confortável em usar o nome de um array como sendo o ponteiro para o item de índice 0 (zero), então pode sempre pegar o endereço do item zero explicitamente. É a mesma coisa.

O fato de que o operador '[' seja uma notação de ponteiros escondida permite um uso “esquisito” da notação de arrays graças à propriedade comutativa da adição... Já que 'x+3' é a mesma coisa que '3+x', os dois usos abaixo são exatamente idênticos:

```
y = x[3]; /* 'x' é a base e '3' é o deslocamento. /
y = 3[x]; /* '3', agora, é a base e 'x' é o deslocamento. */
```

É pouco provável que você obtenha algum erro de compilação. Talvez um aviso... talvez...

## ***Declarando e inicializando arrays, estruturas e unions***

Neste tópico quero deixar registrado um comportamento útil, nos compiladores C, que está

especificado e, portanto, é garantido que sempre funcione. Ao declarar e definir apenas um item de um array, estrutura ou union, o compilador automaticamente preencherá o restante dos itens (não inicializados no código fonte) com zeros:

```
/* test.c */
#include <stdio.h>

struct mystruct_s {
    int x;
    int y;
    int z;
};

struct mystruct_s s = { 1 }; /* define apenas 'x'. */
int a[3] = { 2 };           /* define apenas a[0]. */

int main(int argc, char *argv[])
{
    printf("s = { %d, %d, %d};\n"
           "a = [ %d, %d, %d];\n",
           s.x, s.y, s.z,
           a[0], a[1], a[2]);

    return 0;
}
-----%<---- corte aqui ---%<-----
$ gcc -o test test.c
$ ./test
s = {1, 0, 0};
a = [2, 0, 0];
```

Viu só? Definimos apenas o primeiro item da estrutura declarada como 's' e o primeiro item do array 'a', mas o compilador automaticamente zerou os itens restantes.

Ainda, no caso do GCC, existe uma extensão para inicializações de estruturas e unions que pode ser útil. Podemos explicitar qual membro será inicializado. Substitua a inicialização da estrutura 's', acima, por:

```
struct mystruct_s s = { .z = 1 };
```

E você obterá “s = { 0, 0, 1 }”. A mesma coisa pode ser feita com um array, basta explicitar o índice do item a ser inicializado:

```
int a[3] = { [1] = 2 }; /* Inicializa o item [1] com 2 e o resto com zeros.
Índices de arrays sempre começam com [0]. */
```

Esses recursos são extensões que estão disponíveis na especificação C99, mas especificações mais antigas (C89, por exemplo) não as implementam.

Outra maneira de atribuir valores a um array é usando “literais compostos”. Esse é um recurso que foi introduzido como extensão no GCC e tornou-se padronizado na especificação ISO C11. Confesso que é um “macete” que não costumo usar e até acho meio esquisito. Consiste em informar o array literal precedido por um *type casting*. Assim:

```
double *p = (double []){ 1, 2, 3 };
```

O casting é necessário por que o compilador pode ficar confuso se ele não for informado. Outra consideração útil ao usar esse “macete” é que criamos um ponteiro para um array constante. Isso é diferente de declarar:

```
double p[] = { 1, 2, 3 };
```

Isto é, usar literais compostos podem causar o mesmo problema que temos ao declarar strings contantes atribuídas a ponteiros, ao invés de arrays.

Os literais compostos podem ser usados para passar arrays literais para funções:

```
extern int f(int *x);  
...  
x = f((int []){ 1, 2, 3});
```

De qualquer maneira, acho essa extensão muito esquisita...

## Ponteiros, funções e a pilha

Ponteiros são muito úteis quando trata-se de passar parâmetros para funções “por referência”, ao invés de “por valor”. Especialmente se estamos falando de estruturas complexas. Suponha que tenhamos uma estrutura com vários itens, como:

```
struct vectice_s {  
    float x, y, z, w;  
    float nx, ny, nz;  
    float r, g, b, a;  
    float s, q;  
};
```

Essa estrutura contém 13 *floats* e tem o tamanho total de 52 bytes (cada *float* tem 4 bytes de tamanho). Agora, repare nas declarações de funções abaixo:

```
extern int CheckNormal(struct vectice_s v);  
extern int CheckNormal2(struct vectice_s *vp);
```

Se chamarmos a primeira função todos os 52 bytes da estrutura serão empilhados antes da chamada. De fato, 56 bytes da pilha serão usados, já que na arquitetura x86-64 todos os itens na pilha devem ser alinhados por QWORDs (de 8 em 8 bytes).

Se usarmos a segunda função, tudo o que será passado para a função *CheckNormal2* serão os 8 bytes que correspondem ao tamanho de um ponteiro. Esses 8 bytes podem nem mesmo serem empilhados, de acordo com a *convenção de chamada* que mostrarei no próximo capítulo. Nas declarações acima, o primeiro caso é um exemplo de passagem por valor. O segundo, passagem por referência.

Passagem por valor, especialmente de estruturas complexas, pode aumentar a pressão sobre a pilha. Na linguagem C é muito comum dividir o trabalho em funções e realizar diversas chamadas... Se para cada função chamada empilharmos, digamos, 56 bytes, e tivermos 50 chamadas acumuladas, o uso da pilha será de, aproximadamente, 32 KiB (32000 bytes, exatamente, além dos 56 bytes de parâmetros empilhados temos que contar também com o endereço de retorno das funções chamadoras!). Não parece grande coisa, não é? Acontece que 32 KiB é o tamanho de todo o cache L1 para cada núcleo, em certas arquiteturas. E, num ambiente multithreaded, podemos ter pilhas menores que isso. Quero dizer que, com uso intenso da pilha, teremos problemas em outras áreas...

A pilha é usada para conter outras coisas além de parâmetros de funções e endereços de retorno. Não estamos contando aqui com variáveis locais que não puderam ser mantidas em registradores, bem como variáveis temporárias e valores de registradores que precisam ser preservados entre chamadas... Ou seja, se usarmos passagem de parâmetros por valor de estruturas complexas, num programa grande, corremos o risco de exaurir a pilha<sup>25</sup>, tomando um erro de “stack overflow” na cara, e abortar o programa no meio de um processamento crítico.

---

<sup>25</sup> Mas não é necessário preocupar-se demais com a pilha... Os sistemas operacionais modernos alocam bastante espaço para elas no *userspace*. Cerca de 1 MiB é alocado para ela... E a pilha ainda pode crescer. Só que isso vale para a thread principal do processo. As threads secundárias normalmente são inicializadas com uma pilha de tamanho fixo e pequeno.

## Entendendo algumas declarações “malucas” usando ponteiros

Se não bastasse todas as explicações acima, que afugentam os novatos, temos o verdadeiro terror na flexibilidade e na sintaxe usada por C e C++ com a declaração de ponteiros. Afinal, você pode querer um ponteiro simples para um tipo primitivo ou pode querer algo como “um ponteiro para um array de ponteiros de funções”. Comecemos por algo simples. Declarar um array de ponteiros:

```
int *iArray[3]; /* array de 3 ponteiros do tipo int. */
```

Mas, e se quiséssemos declarar um “ponteiro para um array de 3 *ints*”? Notou a diferença? No caso acima temos 3 ponteiros num array e agora quero um ponteiro para um array de 3 itens. Para obter isso preciso “enganar” o compilador, usando parênteses:

```
int (*iArray)[3]; /* 'iArray' é um ponteiro para um array de 3 ints. */
```

Graças aos parênteses dizemos ao compilador que *iArray* é o ponteiro para um array. No caso anterior o 'int \*' é o tipo do array, no segundo, 'int'.

Eis outro exemplo, no caso de declaração de funções:

```
int *f(void); /* função que retorna ponteiro do tipo int. */
int (*f)(void); /* Ponteiro (não inicializado) para uma função que retorna int. */
```

Outro tipo de declaração que você pode precisar é a de “ponteiro para ponteiro”:

```
char **pp; /* 'pp' é um ponteiro que aponta para um ponteiro que aponta para um char. */
```

Você pode ver a utilidade disso quando temos um array de strings:

```
/* astr é um array de ponteiros. */
char *astr[] = { "hello", "world", NULL };
char **pp;

/* 'pp' aponta para o primeiro item do array acima. */
pp = astr;

/* Percorre o array 'astr' usando o ponteiro 'pp'. */
while (*pp != NULL)
{
    printf("%s\n", *pp);
    printf("O primeiro char da string é '%c'\n", **pp);
    pp++;
}
```

Aqui faço 'pp' apontar para o primeiro item do array 'astr', composto de 3 ponteiros para *char*. '\*pp' nos dá, então, o endereço inicial de cada string, individualmente. Se usássemos '\*\*pp', obteríamos o primeiro caractere da string apontada por '\*pp'.

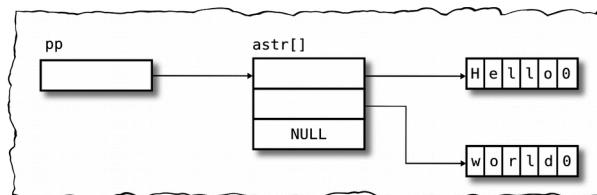


Figura 7: Estrutura usando ponteiro para ponteiros.

A utilidade desse tipo de construção vem do fato que se um ponteiro pode ser usado para apontar para um item de um array, então se usarmos um array de ponteiros podemos ter uma estrutura bidimensional ou um “array de arrays”<sup>26</sup>.

26 Isso é bem diferente de um “array bidimensional” no contexto da linguagem C. O que chamo de “array de arrays” é um “ponteiro para um array de ponteiros”. Há uma distinção sutil.

O que acabo de descrever é o que ocorre com o segundo parâmetro da função *main*. Quando você executa seu programa, a linha de comando é dividida em strings (cada string é um array de chars) e os ponteiros para essas strings são colocados num array. A função *main* recebe o ponteiro para esse array de ponteiros. É por isso que declarar 'argv' como 'char \*argv[]' ou 'char \*\*argv' é, essencialmente, a mesma coisa:

```
int main(int argc, char *argv[]); /* Esta é a forma canônica da declaração de 'main' */
int main(int argc, char **argv); /* Essa é a mesma coisa que a declaração acima. */
```

Com relação a outras declarações “malucas”, podemos ter algumas bem complicadas:

```
int * (*f[10])(int); /* array de 10 ponteiros para funções que retornam ponteiros para int. */
```

Pode ser desafiante criar declarações para, por exemplo, um “array de ponteiros de funções que retornam ponteiros para arrays de funções”... Recomendo que você evite esses tipos de maluquices.

## **Utilidade de ponteiros para funções**

Uma dúvida que um leitor me apresentou foi sobre os ponteiros para funções. Acho que ficou claro que os compiladores usam apelidos para endereços de coisas que estão localizadas na memória. Funções não são diferentes.

Em assembly uma chamada para um procedimento, via instrução CALL, usa como parâmetro o endereço de entrada do procedimento. E como “ponteiro” é apenas uma forma mais rebuscada de falarmos de “endereço”, podemos dizer que CALL usa um ponteiro. Em C podemos declarar um ponteiro para uma função e chamá-la através deste ponteiro:

```
/* funcptr é declarada como um ponteiro para uma função. */
int (*funcptr)(int);

/* Esta é, de fato, a definição de uma função. */
int f(int x) { return x + x; }

int main(int argc, char *argv[])
{
    /* Atribui ao ponteiro 'funcptr' o endereço do
       ponto de entrada da função 'f'. */
    funcptr = f;

    /* Chama a função 'f' indiretamente, usando o ponteiro 'funcptr'. */
    printf("O dobro de %d é %d\n", 2, funcptr(2));

    return 0;
}
```

O próprio símbolo 'f' é um ponteiro, se você pensar bem... ele é o endereço do ponto de entrada da função.

Qual é a utilidade desse artifício? Suponha que você tenha duas ou mais funções que fazem a mesma coisa, mas que são otimizadas para processadores diferentes (ou, usando extensões diferentes do processador). Por exemplo, três funções que preencham buffers com zero... uma escrita em C puro, uma que use SSE e outra escrita em assembly puro. No seu código você quer chamar uma delas, através da mesma chamada, de acordo com uma escolha feita na inicialização da aplicação:

```
/* Protótipos para as 3 funções disponíveis. */
extern void zerofill_c(void *, size_t);
extern void zerofill_sse(void *, size_t);
extern void zerofill_asm(void , size_t);

/* Ponteiro para uma função. */
void (*zerofill_ptr)(void *, size_t);
```

```

/* Rotina de inicialização chamada no início do seu código. */
void init(void)
{
    if (sse_present)
        zerofill_ptr = zerofill_sse;
    else if (can_use_asm)
        zerofill_ptr = zerofill_asm;
    else
        zerofill_ptr = zerofill_c;
}

int main(int argc, char *argv[])
{
    ...
    /* init vai decidir qual das 3 funções usar. */
    init();
    ...

    /* Chama zerofill_xxx Via ponteiro! */
    zerofill_ptr(buffer, sizeof(buffer));

    ...
    return 0;
}

```

## A biblioteca padrão: *libc*

Toda vez que você compila e “linka” um código em C, leva junto símbolos da a biblioteca padrão *libc*. No Linux toda biblioteca começa com o nome *lib*, seguida do que ela contém. “*libc*” contém, além das funções da biblioteca padrão, as rotinas de inicialização e finalização de seus programas.

Diferente do que você pode ter aprendido, a função *main* não é a primeira coisa a ser executada no seu código. A primeira função executada chama-se *\_start*. O diagrama abaixo mostra algumas das funções que são chamadas antes (e depois) de *main*:

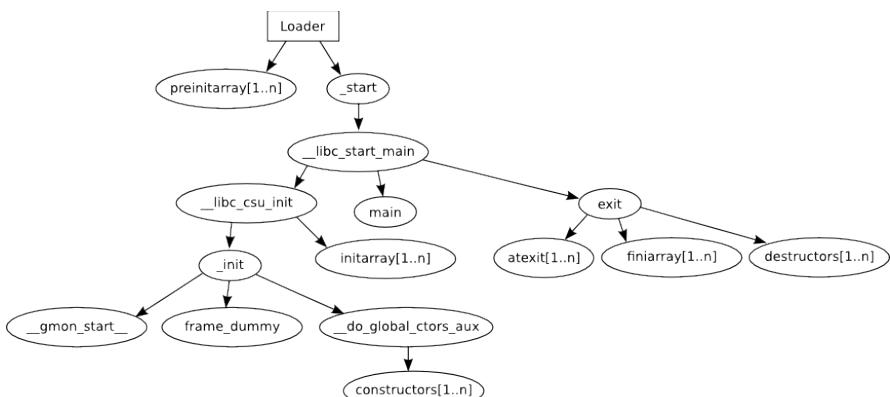


Figura 8: Inicialização de um programa em C

Procure interpretar esse gráfico de chamadas como uma árvore binária: *\_start* chama *\_\_lib\_start\_main*, que chama *\_\_libc\_csu\_init* e depois *main* e depois, na ordem, *\_init*, *gmon\_start*, *frame\_dummy*, *do\_global\_ctors\_aux* e cada um dos construtores registrados no array *constructors[1..n]*. Depois dessas inicializações todas a função *main* é finalmente chamada. Quando ela sai, *exit* é executado e ele executa todos as funções registradas com *atexit*, depois as registradas no array *finiarray* e os destrutores...

Só para ilustrar, eis o código fonte completo de um *helloworld*, totalmente escrito em assembly, que pode ser executado e não usa a *libc*<sup>27</sup>:

27 O padrão de chamada para syscalls no modo 32 bits é um pouco diferente do modo 64 bits. Em ambos os modos poderíamos usar a instrução ‘int 0x80’, mas a instrução syscall é mais indicada. Ao usar syscall a parametrização das funções é diferente do que ao usar ‘int 0x80’.

```

; helloworld.asm
bits 64
section .data

msg: db "Hello, world!", 10
len equ $ - msg

section .text

global _start
_start:
    mov rax,1      ; sys_write syscall
    mov rdi,1      ; STDOUT
    mov rsi,msg
    mov rdx,len
    syscall

    mov eax,60      ; sys_exit syscall
    xor rdi,rdi    ; código de erro de saída.
    syscall
-----%<---- corte aqui -----%>-----
$ nasm -f elf64 helloworld.asm -o helloworld.o
$ ld -s helloworld.o -o helloworld
$ ./helloworld
Hello, world!
$ ls -l helloworld
-rwxrwxr-x 1 user user 512 Dez 29 21:43 helloworld

```

O símbolo `_start` é usado pelo linker, por default, como ponto de partida para o executável. Não significa que todo executável tem que ter esse símbolo. Poderíamos usar a opção '`-e`' do linker para estipular um ponto de entrada diferente:

```
$ ld -e _mystart -s myprog.o -o myprog
```

Nos programas em C a libc é o motivo pelo qual seu código não terá menos que 8 KiB de tamanho. Repare que a versão em puro assembly tem apenas 512 bytes. A biblioteca padrão faz muito trabalho de inicialização e finalização.

De uma forma geral, a função `_libc_csu_init` é responsável pela execução de *construtores*... Eles não são coisas restritas ao C++. Graças ao esquema de atributos do GCC podemos criar funções que serão executadas **antes** de `main`, como se fossem construtores de objetos, no C++. Da mesma forma, existem funções de finalização, ou “destrutores” que são chamados quando o código em C é encerrado:

```

/* test.c */
#include <stdio.h>

void __attribute__((constructor)) ctor(void)
{
    printf("Função chamada antes de main().\n");
}

void __attribute__((destructor)) dtor(void)
{
    printf("Função chamada depois de main().\n");
}

int main(int argc, char *argv[])
{
    printf("Hello!\n");
    return 0;
}
-----%<---- corte aqui -----%>-----
$ gcc -o test test.c
$ ./test
Função chamada antes de main().
Hello!
Função chamada depois de main().

```

O código de inicialização é a porção da *libc* que é linkada de forma estática. O restante das funções, como *printf*, por exemplo, são “linkadas” de forma dinâmica. O GCC assume a opção “-lc” por default<sup>28</sup>.

*A libc faz um monte de coisas “por baixo dos panos”*

Eis uma comparação com o código *helloworld.asm*, mostrado anteriormente, e o clássico *hello.c*. Para compará-los vou usar a ferramenta *strace* que mostra apenas chamadas *syscalls*. Se você quiser ver, inclusive, chamadas à bibliotecas, use o utilitário *ltrace* com a opção '*-S*':

Pera ai! Porque `execve` está sendo chamada? Acontece que *Linux* não cria um processo “do nada”. Todos os processos são *forks* do processo *init* e depois é feita a chamada para `execve` para substituir o “novo” processo pela imagem contida no arquivo executável...

Depois disso, a primeira coisa que você nota é a tentativa de abrir um arquivo de configuração

<sup>28</sup> A opção `-l` precisa apenas da porção do nome da biblioteca que segue `'lib'` e precede `'.so'` ou `'.a'`. Por exemplo, se você precisar usar funções da biblioteca `libORbit-2.so`, só precisará especificar `"-lORbit-2"` para o GCC.

chamado *ld.so.nohwcap*. Isso existe porque algumas bibliotecas podem ter sido pré-carregadas e estão atreladas a features específicas do seu processador. Se */etc/ld.so.nohwcap* existir e seu conteúdo for diferente de '0', então a libc vai tomar alguns cuidados ao carregar essas bibliotecas especiais.

A seguir, a libc consulta a configuração de *ld.so.preload* para que ela não tente carregar as bibliotecas listadas nesse arquivo de configuração (já que elas já estão carregadas!).

O arquivo *ld.so.cache* contém bibliotecas candidatas a já estarem carregadas e prontas para ser usadas. A libc verifica isso também...

Durante a carga uma série de alocações de redimensionamentos de páginas privadas ao processo são feitas e a biblioteca *libc.so* é carregada. Até então o que foi executado eram *syscalls*.

A chamada a *arch\_prctl* inicializa o seletor FS para o processo (possivelmente usado durante chaveamento de contextos de tarefas).

Finalmente, depois de mais alguns ajustes a syscall *write* é chamada para imprimir nossa string e *exit* é chamada.

Esse trabalho todo é um dos motivos pelos quais desenvolver aplicações inteiras em assembly é impraticável.... A *libc* contém um enorme conjunto de *features* que facilitam muito nossas vidas. Sem ela você teria que criar tudo isso manualmente, à medida que necessitar.

A vantagem da *libc* é que essas inicializações são feitas apenas uma vez, bem como rotinas de finalização – se necessárias. Uma vez que o código em *main* é executado, todo o controle está com a sua aplicação. É claro, de tempos em tempos você cede esse controle à *libc* ou a outras bibliotecas, como *pthread*, por exemplo. Mas é um preço muito pequeno para pagar em troca de performance, estabilidade e disponibilidade de ferramentas interessantes.

## Cuidados ao usar funções da *libc*

As funções da *libc* são construídas para funcionar de acordo com certas condições. Por exemplo, as funções *strtok*, *setlocale*, *fcloseall*, *readdir*, *tmpnam* e *gethostbyname* não são seguras para serem usadas em threads. Elas são marcadas como *MT-Unsafe* na documentação da biblioteca (pena que isso não seja explícito nas *manpages*). Além da insegurança quando a multithreading, muitas funções não são seguras para serem usadas em tratadores de sinais ou, pior, não são seguras para serem interrompidas por tratadores de sinais. Essas são marcadas como *AS-Unsafe* (de *assynchronous signal*).

Existem ainda aquelas que não são seguras para serem usadas em tratadores de cancelamento de threads. Essas são marcadas como *AC-Unsafe* (de *asynchronous cancelation*).

Consulte o manual da GNU *libc*<sup>29</sup> para maiores detalhes.

## Disparando processos filhos

Outra dúvida frequência a respeito do Linux é como fazer para que seu programa execute outro programa. No Windows temos a função *CreateProcess* e, parece, existe algo semelhante no Linux, que faz parte da biblioteca padrão (*libc*): As funções *exec*.

Antes de falar de *exec* preciso explicar que existe uma função genérica chamada *system* **que não deve ser usada!** Essa função é bem simples: Você passa uma string contendo a linha de comando desejada e executa a função. O valor de retorno é o código de retorno da linha de comando ou -1,

---

<sup>29</sup> Baixe o pdf neste link: <http://www.gnu.org/software/libc/manual/pdf/libc.pdf>

em caso de erro<sup>30</sup>. Acontece que system tem o potencial de causar graves problemas de segurança e não é recomendado por diversos *advisories* especializados. O método correto de disparar um novo processo é através do par de funções *fork* e *exec*.

*Forking* é a ação de criar uma cópia do processo atual de forma que ambos os processos continuarão a execução a partir do *fork*. A função *fork* retornará um de três valores:

- -1, se o novo processo não pode ser criado a partir do processo atual;
- 0 é retornado para o processo filho;
- Um valor positivo é retornado para o processo pai, contendo o PID do processo filho.

A função *fork* tem um efeito colateral interessante: O processo filho herda todos os dados, *descritores de arquivos*, e outros recursos do pai. Esses recursos só divergirão do pai quando forem modificados pelo filho.

A partir da criação do processo filho com base no processo pai, queremos que esse novo processo seja **substituído** pela carga de um novo executável. Isso é feito por *exec*. Ele sobrepõe o processo. O código para executar um processo filho arbitrário, dessa maneira, é esse:

```
static char const * const arg = "./proc2";
pid_t pid;

...
if ((pid = fork()) == -1)
{
    fprintf(stderr, "ERRO: Não foi possível carregar o processo filho!");
    exit(EXIT_FAILURE); /* Ou outra instrução para desistir da criação do processo filho. */
}

/* Se for o filho, substitui o processo carregando um novo executável.
Note que o primeiro parâmetro é igual ao segundo. O primeiro parâmetro é
o arquivo que será "executado". O segundo parâmetro é equivalente a argv[0] e
assim por diante. */
if (pid == 0)
    execl(arg, arg, NULL);

/* ... o pai continua aqui ... */
...
```

Mesmo depois da execução de *execl* (ou derivados), alguns atributos do processo original são preservados, afinal o processo ainda é um *fork* do processo original, só com a imagem binária modificada. Mas a maioria desses atributos são substituídos, ou seja, são descartados. Como o mapeamento de memória, os dispatches e máscaras de *signals*. Em essência, é como se todos os atributos fossem recriados para a nova imagem binária. Consulte a documentação de *execve* para mais detalhes.

---

30 Embora existam diferenças entre sistemas POSIX! No Linux, por exemplo, é recomendável que se use o macro WEXITSTATUS para obter o valor real de retorno.



# Capítulo 4: Resolvendo dúvidas sobre a linguagem Assembly

É sempre bom ter em mente que “linguagem de máquina” é a única linguagem que seu processador entende. Todas as outras existem para que você, programador, tenha condições de dizer ao computador o que ele deve fazer.

Essas outras linguagens precisam ser traduzidas e, no processo de tradução, alguma intenção pode ser perdida, mal interpretada.

Com assembly nos aproximamos bastante da “linguagem de máquina” entendida pelo processador.

## O processador pode ter “bugs”

Encare seu processador como se fosse um pequeno computador que está executando, constantemente, um programa. Esse programa é feito para decodificar instruções e executá-las. Só que, como em qualquer programa, ele pode conter bugs.

Para saber quais bugs seu processador tem é necessário saber qual é a versão só “software” que ele está rodando. Isso é feito fazendo uma consulta via instrução CPUID:

```
/* version.c */
#include <stdio.h>
#include <string.h>
#include <cpuid.h>

struct version_s {
    unsigned stepping:4;
    unsigned model:4;
    unsigned family:4;
    unsigned type:2;
    unsigned :2;
    unsigned exmodel:4;
    unsigned exfamily:8;
};

static const char *processor_types[] =
{ "Original OEM", "Overdrive", "Dual", "Reserved" };

const char *getBrandString(void);

void main(void)
{
    unsigned a, b, c, d;
    int type, family, family2, model, stepping;

    cpuid(1, a, b, c, d);
    type = ((struct version_s *)&a)->type;
    family = family2 = ((struct version_s *)&a)->family;
    model = ((struct version_s *)&a)->model;
    stepping = ((struct version_s *)&a)->stepping;

    if (family == 6 || family == 15)
    {
        if (family == 15)
            family2 += ((struct version_s *)&a)->exfamily;
        model += ((struct version_s *)&a)->exmodel << 4;
    }
}
```

```

printf("Seu processador: \"%s\"\n"
      "\tTipo: %s\n"
      "\tFamília: 0x%02X\n"
      "\tModelo: 0x%02X\n"
      "\tStepping: 0x%02X\n",
      getBrandString(),
      processor_types[type],
      family2, model, stepping);
}

const char *getBrandString(void)
{
    static char str[49];
    unsigned *p;
    size_t idx;
    unsigned a, b, c, d;
    int i;

    p = (unsigned *)str;
    for (i = 2; i <= 4; i++)
    {
        __cpuid(0x80000000 + i, a, b, c, d);
        *p++ = a;
        *p++ = b;
        *p++ = c;
        *p++ = d;
    }
    *(char *)p = '\0';

    /* Retorna string a partir do ponto onde não há espaços. */
    return (const char *)str + strspn(str, " ");
}
-----%<---- corte aqui -----%<-----
$ gcc -o version version.c
$ ./version
Seu processador: "Intel(R) Core(TM) i5-3570 CPU @ 3.40GHz"
    Tipo: Original OEM
    Família: 0x06
    Modelo: 0x3A
    Stepping: 0x09

```

Encare os campos família, modelo e *stepping* como se fosse a versão do software do seu processador. No exemplo acima, em uma de minhas máquinas de teste, temos um i5 cuja versão é 6.58.9. Se você tiver o mesmo “brand” de processador (i5-3570) pode ser que seu *stepping* seja diferente. Para a versão 6.58, se o stepping for maior que 9 então o seu processador tem menos bugs que o meu...

## A mesma instrução pode gastar mais tempo do que deveria

Tomemos o exemplo de uma instrução muito simples:

```
add r/m64,imm64
```

“r/m64” significa que o operando do lado esquerdo aceita um registrador (r) ou referência à memória (m) de 64 bits. O operando do lado direito, “imm64”, diz que essa instrução aceita um valor imediato, um valor.

O manual de otimização da Intel nos diz que essa instrução tem latência de 1 ciclo de máquina, mas isso depende se estamos usando registrador ou memória do lado esquerdo. As duas instruções abaixo tem latência diferentes:

```
add rax,1          ; Gasta 1 ciclo
add qword [rdx],1  ; Gasta 2 ciclos
```

Quando usamos referência à memória um ciclo adicional é gasto porque o valor de 64 bits têm que ser lido, somado ao valor 1 e depois gravado de volta. Isso não acontece quando o operando é um

registrador.

Existem, ainda, algumas idiossincrasias... As instruções abaixo consomem tempos diferentes, mesmo que não haja o ciclo de leitura-operação-escrita:

```
cmp rax,[rsi]      ; Gasta 2 ciclos.  
cmp [rsi],rax     ; Gasta 3 ciclos.
```

Ambas as instruções comparam (subtraem) o conteúdo de um registrador (RAX) com o conteúdo da memória (apontada por RSI). Mas, já que a segunda instrução faz referência no primeiro operando, ela gasta 1 ciclo extra.

Para ter uma ideia intuitiva sobre o gasto de tempo de uma rotina, é bom assumir essa regra: Referências à memória adicionam pelo menos um ciclo de máquina e, se for do lado “destino” da instrução, mais um.

### **Nem todas as instruções “gastam” tempo**

Esse é um conceito difícil de aceitar. Mas é verdadeiro. Algumas instruções são executadas de tal maneira, em certas circunstâncias, que elas não “gastam” tempo algum.

Há algum tempo os processadores podem executar duas ou mais instruções ao mesmo tempo (e não estou falando de “threads” aqui!). Se uma instrução “gasta” 10 ciclos e outra gasta 1 ciclo, a última parece não ter gastado tempo algum. Se a sequência abaixo for executada em paralelo:

```
; As duas instruções gastam 10 ciclos, no total!  
imul ebx      ; Gasta uns 10 ciclos.  
mov ecx,esi    ; Gasta só 1 ciclo.
```

O gasto de tempo da segunda instrução está incorporado no gasto da primeira. Especialmente porque ela não depende de nada da primeira... No exemplo, IMUL multiplica EAX por EBX e coloca o resultado no par de registradores EDX:EAX. Note que o MOV que segue usa ECX e ESI.

Outro exemplo de MOV que não gasta tempo é o acesso a uma região da memória mapeada para o *Local APIC*. Segundo a documentação da Intel, essa região é mapeada internamente e não consome nenhum ciclo de leitura por parte do processador... Esses MOVs podem ser considerados como instantâneos.

### **A pilha e a “red zone”**

Você verá, no próximo capítulo, que variáveis locais e temporárias, usadas por suas funções em C, podem ser “reservadas” na pilha. Mostrarei como isso funciona por lá, mas é necessário explicar um conceito, usado no modo x86-64, chamado de “zona vermelha” (*red zone*).

Esse conceito só existe no modo x86-64 e é definido no POSIX ABI... Ele afirma que os 128 bytes **depois** do topo da pilha (depois que foi alocado espaço para as variáveis locais) **não** pode ser usado pelo kernel. Essa regra é útil porque o kernel usa a pilha do processo para manter informações sobre a thread a qual ele pertence.

Ao criar essa zona “desmilitarizada”, digamos assim, o kernel dá um espaço de trabalho confortável para a thread do processo e permite que fragmentos de código chamados *prólogo* e *epílogo* sejam desnecessários...

No modo x86-64 é possível compilar o kernel para que ele não respeite a “red zone”, mas isso tem o potencial de tornar o sistema operacional instável, no que concerne às threads...

## Prefixos

Algumas instruções, por causa do uso dos registradores, são codificadas com tamanho maior do que você espera. Eis um exemplo:

```
00000000: 31C0 xor eax,eax  
00000002: 4831C0 xor rax,rax
```

Esse byte 0x48 na frente dos dois bytes que compõem um “XOR EAX,EAX” é o prefixo REX. Ele indica que a instrução usa registradores estendidos. No exemplo acima as duas instruções fazem exatamente a mesma coisa: zeram RAX.

Outros prefixos existem. Algumas instruções especiais possuem o prefixo 0x0F. Outras são prefixos de repetição, como REP, REPZ e REPNZ:

```
00000000: A5 movsd  
00000001: F3A5 rep movsd  
00000003: F3A5 repnz movsd  
00000005: F2A5 repz movsd
```

O prefixo REPNZ é o mesmo usado pelo REP (são a mesma coisa). Já o REPZ é um prefixo diferente. Além do prefixo temos a instrução MOVSD (0xA5).

Existem mais prefixos. Se sua instrução usa um ponteiro e um seletor de segmento diferente de DS, ela será prefixada, indicando o seletor usado. O mesmo vale quando usar um seletor diferente de SS para ponteiros com endereço-base usando RSP ou RBP.

Além desses prefixos corriqueiros, algumas extensões do processador usam prefixos especiais. Existe um prefixo XOP (*eXtended OPeration*), disponível para extensões AVX. E existem extensões que são “dicas” para o processador, raramente usadas. Mas, dois prefixos especiais podem ser muito frequentes: 0x66 e 0x67.

O processador espera encontrar operandos de 32 bits, nos modos i386 e x86-64. Se você usar operandos de 16 bits, como os registradores AX, BX, CX, ..., ele colocará o prefixo 0x66 (operand override):

```
00000000: 66678B06 mov ax,[esi]  
00000004: 678B06 mov eax,[esi]
```

Isso significa que a maioria das instruções que lidam com o tipo *short* podem ter um byte a mais e, por esse motivo, o compilador prefere lidar com *ints*, nem que seja apenas no código final.

O prefixo 0x67 é o *address override*. No modo x86-64, sempre que temos uma indireção como [ESI], o processador prefere usar registradores de 64 bits, afinal um endereço canônico tem 52 bits de tamanho. Daí, se usássemos [RSI] na instrução acima o prefixo 0x67 não seria usado:

```
00000000: 8B06 mov eax,[rsi] ; Operandos e endereços do tamanho esperados.  
00000002: 678B06 mov eax,[esi] ; Operando de 32 bits, mas endereço em 32 bits!  
00000005: 668B06 mov ax,[rsi] ; Operando de 16 bits, mas endereço em 64!  
00000008: 66678B06 mov ax,[esi] ; Operando de 16 e endereço de 32!  
0000000C: 488B06 mov rax,[rsi] ; Operando de 64 e endereço de 64 (prefixo REX).  
0000000F: 67488B06 mov rax,[esi] ; Operando de 64 (REX) e endereço de 32 (0x67).
```

O importante é perceber que se seus operandos foram de 32 bits e os ponteiros de 64, nenhum prefixo é adicionado à instrução.

E, por estranho que pareça, o uso de operandos de 8 bits não acrescenta prefixos, a não ser que o outro operando seja uma indireção usando ponteiros de 32 bits:

```
00000000: 8A06 mov al,[rsi]
```

**Atenção!** O prefixo REX não é fixado em 0x48. De fato, existe mais que um desses prefixos,

dependendo do formato da instrução. Mas, os prefixos 0x66 e 0x67 são fixos e suas semânticas são as citadas.

A dica aqui é a de que você deve evitar o uso de tipos de 16 ou 64 bits tanto quanto possível, mesmo no modo x86-64, para evitar a adição de prefixes REX ou 0x66. O prefixo 0x67 raramente é usado pelo compilador C porque ele respeita o tamanho dos ponteiros. Ao usar um tipo *short* você pode estar poupando espaço no cache L1D, mas estará colocando mais pressão onde mais interessa, no seu código (ou seja, no cache L1I)!

Dito isso você perceberá que, mesmo que use o tipo *short*, o GCC tende a usar registradores de 32 bits através de instruções como MOVZX (para *unsigned short*) e MOVSX (para *signed short*). Isso não poupa espaço em relação a usar um registrador de 16 bits, que terá o código prefixado com 0x66, mas usará um prefixo 0x0F... O prefixo 0x0F é especial. Ele faz parte da instrução e não é lá um “prefixo” e, por isso, não tem o potencial a inserir ciclos na execução. O GCC, ao encontrar o código abaixo, tende a gerar código como segue:

```
short x = *(short *)p;
-----%<---- corte aqui -----%<-----
000000??: 0FB706  movsz eax,word [rsi] ; Assumindo que RSI contém o endereço em p.
```

Como vimos anteriormente, a instrução “MOV AX,[RSI]” tem 3 bytes de tamanho, com um prefixo 0x66. A instrução MOV, neste caso, tende a ser mais lenta que MOVSX.

É claro que nem sempre o GCC usará essa técnica. Por exemplo, o código pode precisar da parte superior de EAX, mas ele tem preferência por fazer isso.



# Capítulo 5: Misturando C e Assembly

Existem duas formas de misturar funções escritas em C e em assembly. No primeiro modo é usando o que se chama de *assembler inline*. “Assembler” (com ‘er’) significa montador. “Inline” porque o código em linguagem **assembly** (com ‘y’) será misturado com o código em C.

O segundo modo é usando um **assembler** externo, como o MASM, GAS ou NASM. Usarei esse último por sua simplicidade e portabilidade (além do que, é “free software”).

Antes de começarmos a bagunça é importante entender como o compilador C organiza as funções para que possamos usar o mesmo padrão com as listagens em assembly, usando um assembler.

## Convenções de chamada (x86-64)

Toda função em C assume uma ordem em que registradores são usados ou dados serão empilhados, formando os parâmetros que a função receberá, bem como quais registradores retornarão valores e quais devem ser preservados entre chamadas. A isso se dá o nome de “convenção de chamada”.

Na arquitetura x86-64 a convenção usada depende do sistema operacional. No Windows é de um jeito e nos sistemas baseados em POSIX (Linux, OS/X, Unix etc) é de outro. No caso dos sistemas baseados em POSIX a *interface binária de aplicações* (ABI, *Applications Binary Interface*) dita que os parâmetros sejam passados para uma função usando os registradores RDI, RSI, RDX, RCX, R8 e R9, nessa sequência. Se houverem valores em ponto flutuante, os registradores XMM0 até XMM7 devem ser usados, também nessa sequência. Parâmetros adicionais são passados pela pilha. Os valores de retorno são colocados no registrador RAX (ou XMM0, no caso de ponto flutuante). Os registradores RBP, RBX e os registradores entre R12 até R15 devem ser sempre preservados entre chamadas.

Com o Windows a coisa não é tão flexível. Os registradores RCX, RDX, R8 e R9 são usados, na sequência, ou XMM0 até XMM3, em caso de ponto flutuante. Os demais parâmetros são passados pela pilha. Da mesma forma que na POSIX ABI, RAX (ou XMM0, no caso de ponto flutuante) deve ser usado como valor de retorno. A preservação de registradores, além dos especificados pela POSIX ABI, também engloba RSI, RDI e os registradores de XMM6 até XMM15.

Arquitetura	Parâmetros		Retorno	Devem ser preservados
POSIX ABI	<b>Inteiros</b>	RDI, RSI, RDX, RCX, R8 e R9	RAX	RBX, RBP, R12 até R15.
	<b>Ponto flutuante</b>	XMM0 até XMM7	XMM0	
Windows	<b>Inteiros</b>	RCX, RDX, R8 e R9	RAX	RBX, RBP, RSI, RDI, R12 até R15. XMM6 até XMM15
	<b>Ponto flutuante</b>	XMM0 até XMM3	XMM0	

Tabela 4: Registradores usados na convenção de chamada x86-64

Se houverem mais parâmetros do que registradores disponíveis, os valores serão empilhados. O empilhamento é feito da direita para a esquerda, seguindo o padrão da linguagem C, por causa da possibilidade de termos um número variável de parâmetros numa função:

```
int f(int x, ...);
```

No protótipo acima o compilador não tem como saber, de antemão, o número de parâmetros. Se chamarmos a função desta maneira:

```
f(1, 2, 3, 4, 5, 6, 7, 8);
```

O compilador colocará os primeiros 6 parâmetros nos registradores, de acordo com a convenção, e os dois restantes na pilha. Empilhando<sup>31</sup> primeiro o valor 8 e depois o 7:

```
mov edi,1  
mov esi,2  
mov edx,3  
mov ecx,4  
mov r8d,5  
mov r9d,6  
sub rsp,16          ; Ajusta RSP para empilhar os dois dwords.  
mov dword ptr [rsp],7      ; Note que 7 está no topo da pilha.  
mov dword ptr [rsp+8],8  
call f  
add rsp,16          ; Limpa a pilha.
```

Outro exemplo útil para melhor compreensão da convenção de chamada x86-64 é quando temos uma função assim:

```
void f(int x, float y, int z);
```

O parâmetro *x* será passado através do registrador RDI, o *y* através de XMM0 e *z* pelo RSI. Isso quer dizer que RSI é o segundo parâmetro inteiro da sequência, mesmo que o segundo parâmetro real seja um *float*.

## Convenções de chamada (i386)

Este livro é dedicado ao uso do modo x86-64, mas é interessante saber como o modo i386 lida com a convenção de chamada... Na verdade, existem pelo menos 3 delas:

Convenção	Descrição
cdecl	Todos os argumentos da função são empilhados da direita para esquerda.
fastcall	Os dois primeiros argumentos são passados pelos registradores ECX e EDX, respectivamente. O restante é empilhado da direita para esquerda, como em <i>cdecl</i> .
stdcall	Variação da antiga convenção <i>pascal</i> . Os argumentos são passados da direita para esquerda, pela pilha. Ainda, a função chamada é responsável por “limpar” a pilha antes de sair.

Tabela 5: Convenções de chamada no modo 32 bits.

Assim como na convenção usada no modo x86-64, o registrador EBX deve sempre ser conservado.

A convenção *cdecl* é a convenção padrão para qualquer compilador C. As outras, se usadas, devem ser explicitamente informadas.

No caso do Windows é comum observar declarações como esta:

```
int CALLBACK WinMain(HINSTANCE, HINSTANCE, LPSTR, int);
```

Esse *CALLBACK* na declaração da função é, certamente um macro que é substituído por *\_stdcall*. Historicamente, todas as funções da API do Windows, exceto uma, seguem a convenção PASCAL

<sup>31</sup> Repare: Mantive a pilha alinhada por *qword*, colocando os valores de 32 bits em espaços de 64. Isso é necessário para manter a performance!

(ou *stdcall*). A exceção é a função *wsprintf*, que aceita número variável de parâmetros... O que só é possível se o empilhamento for feito da direita para esquerda. Assim, as funções da Win32 API esperam que os argumentos sejam empilhados da esquerda para a direita e que a função chamadora “limpe” a pilha.

“Limpar” a pilha significa somente colocar o ESP na posição inicial, antes do empilhamento dos parâmetros e da chamada da função.

Quanto aos valores de retorno, todas as convenções acima usam o registrador EAX e, às vezes, também o registrador EDX. O par EDX:EAX é usado quando estamos lidando com tipos com mais de 64 bits de tamanho.

Para entender a convenção padrão *cdecl*, eis um exemplo:

```
int f(int a, int b) { return a + b; }
```

O código gerado pela função acima pode aparecer de várias formas. A mais provável, se a máxima otimização for usada, é a que vem a seguir:

```
f:    mov  eax,[esp+4]      ; Pega 'a'.
       add  eax,[esp+8]      ; soma com 'b'.
       ret
```

Lembre-se que sempre que um valor é colocado na pilha, o ponteiro de pilha (Stack Pointer) é decrementado e depois o valor é gravado no lugar apontado. Como a função *f* usa a convenção *cdecl*, sabemos que 'b' é empilhado primeiro, seguido por 'a'. Depois dos dois empilhamentos a instrução *call* é usada para chamar a função, o que causa o empilhamento do endereço de retorno à função chamadora. É por isso que usamos [esp+4] para obter o valor de 'a' (o último valor empilhado), já que ESP aponta para uma posição da pilha que contém o endereço de retorno...

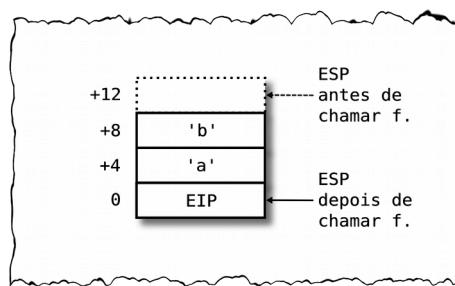


Figura 9: Pilha, antes e depois de chamar a função com a convenção *cdecl*.

Na figura acima, 'a' é o valor contido no argumento *a*, passado para a função. Assim como 'b' é o valor do argumento *b*. A chamada é feita assim:

```
x = f(1,2);
-----%<----- corte aqui -----%<-----
; Código equivalente... note a ordem do empilhamento...
push dword 2 ; Empilha 2
push dword 1 ; Empilha 1
call f
sub esp,8     ; Limpa a pilha.
```

O código em assembly, acima, é apenas um exemplo didático. É mais provável que seu compilador faça algo assim:

```
add esp,8          ; Aloca 2 'ints' na pilha.
mov dword ptr [esp+4],2 ; Coloca os 2 ints na ordem esperada.
mov dword ptr [esp],1
call f            ; chama f.
sub esp,8         ; limpa a pilha.
```

Outros detalhes dignos de nota são:

- O tipo *float* é passado como se fosse um tipo *int*, ou seja, o conteúdo binário da variável é empilhado no mesmo espaço de 32 bits que seria usado por um *int*. **Note** que não há uso de registradores SSE, automaticamente, como na convenção usada pela arquitetura x86-64;
- Tipos com 64 bits de tamanho são passados através do empilhamento de dois valores de 32 bits. A porção mais significativa é empilhada primeiro. O motivo é que ao acessar esses valores via ponteiros, de dentro da função, a ordem *little endian* tem que continuar sendo obedecida;
- Tipos com menos de 32 bits (*char* e *short*) são convertidos para o equivalente de 32 bits antes de serem empilhados;
- Como não há possibilidade de usar o par de registradores RDX:RAX no modo 32 bits, o tipo estendido *\_int128* tende a não estar disponível.

Neste ponto você pode se perguntar: “Por que a convenção *fastcall* não é a preferida dos compiladores no modo i386?”. Afinal, usar registradores nos dois parâmetros iniciais evita, pelo menos, dois empilhamentos! Acontece que *fastcall* não é usado por nenhum sistema operacional e os criadores de compiladores estão livres para implementar essa convenção como bem os convier. É o caso do *Borland C++ Builder* (ou *Embarcadero C++ Builder*, como preferirem!)... Neste compilador há a tendência de usar 3 registradores ao invés de 2: EAX, EDX e ECX, nesta ordem, numa convenção chamada por eles de *register*. Como *fastcall* não é, nem de perto, padronizada, então é prudente evitá-la. A não ser que você saiba realmente o que está fazendo...

De todo modo, a convenção padronizada do modo x86-64 é superior às convenções mostradas aqui. Os pontos de interesse nesse tópico são: A convenção de chamada no modo i386 é **diferente** da usada no modo x86-64 (o que as torna incompatíveis); e, mesmo no modo x86-64, alguns parâmetros podem ser passados pela pilha, do mesmo jeito que na convenção *cdecl*.

## **Funções com número de parâmetros variável no x86-64**

Na convenção *cdecl* os parâmetros são passados pela pilha, mas no x86-64, os 6 primeiros são passados por registradores. Como é que fica o caso de chamadas para funções que possuem número variável de parâmetros? Suponha que tenhamos a declaração de *f*, abaixo, e uma chamada:

```
extern void f(int x, ...);
f(1,2,3,4);
-----%<----- corte aqui -----%<-----
extern f

g:
    mov ecx,4
    mov edx,3
    mov esi,2
    mov edi,1
    call f
```

Quer dizer, a convenção de chamada x86-64 continua valendo, inclusive para funções com número de parâmetros indefinido...

## **Pilha, nos modos i386 e x86-64**

Não sei se ficou evidente. Pilhas, nesses dois modos, comportam-se de maneiras diferentes. No modo i386 cada entrada na pilha tem 32 bits de tamanho. Isso quer dizer que ESP será incrementado ou decrementado de 4 em 4 bytes. Já no modo x86-64, além de usarmos o registrador

estendido RSP, cada entrada na pilha tem 64 bits de tamanho, ou seja, quando RSP for “movido”, será de 8 em 8 bytes.

### **Um detalhe sobre o uso de registradores de 32 e 64 bits**

Na arquitetura x86-64 os registradores de 32 bits são um subconjunto dos registradores de 64 bits. EAX, por exemplo, é a porção dos 32 bits menos significativos do registrador RAX. No caso dos registradores estendidos, R8 até R15, para acessar os 32 bits inferiores basta usar um sufixo: R8D até R15D, onde 'D' significa DWORD. Se usar o sufixo 'W' estaremos acessando os 16 bits menos significativos – 'W' de WORD.

O detalhe que quero mostrar é que, ao usar EAX, ao invés de RAX, os 32 bits superiores do registrador de RAX são automaticamente zerados. Isso se deve ao fato que a instrução que lida com EAX é a mesma que lida com RAX. Por exemplo:

```
; test.asm
bits 64
section .text
global f:function

f:
    mov rax,1
    mov eax,1
    ret
-----%<---- corte aqui -----%<-----
$ nasm -f elf64 -l test.s -o test.o test.asm; cat test.s
1           bits 64
2           section .text
3           global f:function
4
5
6           f:
7 00000000 B801000000      mov rax,1
8 00000005 B801000000      mov eax,1
9 000000A C3               ret
```

Repare que o microcódigo ( $\mu_{op}$ ) das duas instruções MOV são idênticas. Elas só serão diferentes se RAX for inicializado com um valor maior que 32 bits.

Claro que o que se aplica a EAX e RAX, aplica-se a todos os outros registradores de uso geral (RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8 até R15).

### **Exemplo de função em assembly usando a convenção de chamada**

Eis as listagens de uma simples função, em C e o código gerado pelo compilador, em Assembly:

```
/* simple.c */
int f(int x, int y, int z)
{
    return x * y + z;
}
-----%<---- corte aqui -----%<-----
; simple.s - gerado pelo compilador
; Entrada:
;     RDI = x
;     RSI = y
;     RDX = z
; Saída:
;     EAX
f:
    imul edi,esi          ; RDI = EDI*ESI (x = x*y)
    lea   eax,[rdi+rdf]   ; EAX = EDI+EDX (return x+z)
    ret
```

Ao ser chamada pelo código em C, a função 'f' receberá o valor de 'x' pelo registrador RDI, 'y' pelo

RSI e 'z' pelo RDX, de acordo com a convenção. RAX é usado para devolver o resultado que, no caso, é do tipo 'int'. Então a rotina preocupa-se apenas com a DWORD menos significativa de RAX (ou seja, EAX). O compilador usará EDI, ESI e EAX porque especificamos o tipo 'int'.

Visual C++ usará a convenção da Microsoft, substituindo RDI por RCX, RSI por RDX e o RDX por R8:

```
...
f proc near
    imul ecx, edx
    lea    eax, [rcx+r8]
    ret
f endp
...
```

### Aviso sobre retorno de funções com tipos complexos

Funções podem retornar tipos complexos como *structs* e *unions*, mas não é uma boa idéia retornar esses tipos por valor... Considere o que acontece quando retornamos valores de tipos primitivos como *int*, por exemplo...

O valor retornado é uma cópia do valor contido dentro da função. No código:

```
int f(int x)
{
    int temp = x + x;
    return temp;
}
```

**Não** estamos retornando a variável *temp*, mas uma cópia de seu conteúdo, para o chamador. A mesma coisa acontece com estruturas retornadas por valor, como no exemplo:

```
struct myStruc {
    int count;
    int array[5];
};

struct myStruc f(int x)
{
    struct myStruc ms = { x, { 0, 1, 2, 3, 4 } };

    return ms;
}
-----%<---- corte aqui -----%<-----
f:
    mov rax,rdi
    mov [rdi],esi      ; ms.count = x;
    mov [rdi+4],0
    mov [rdi+8],1
    mov [rdi+12],2
    mov [rdi+16],3
    mov [rdi+20],4
    ret
```

Repare: A função continua retornando RAX, mas, dessa vez, retorna um ponteiro para a estrutura que deverá ter sido alocada pela função chamadora de *f*. Um ponteiro para essa estrutura é passado pelo registrador RDI, da mesma forma que seria feito se a função fosse escrita assim:

```

struct myStruc *f(struct myStruc *p, int x)
{
    p->count = x;
    p->array[0] = 0;
    p->array[1] = 1;
    p->array[2] = 2;
    p->array[3] = 3;
    p->array[4] = 4;
    return p;
}

```

Este parece ser um recurso interessante, mas dê uma olhada no código gerado pela rotina abaixo:

```

struct myStruc {
    int count;
    int x[512];
};

struct myStruc f(int x)
{
    int i;
    struct myStruc ms = { x };

    for (i = 0; i < 512; i++)
        ms.x[i] = x+1;

    return ms;
}
-----%<---- corte aqui -----%<-----
f:
    push rbp
    mov edx,2052
    mov ebp,esi          ; Guarda o parâmetro x.
    push rbx
    xor esi,esi          ; Vai preencher a estrutura com zeros.
    mov rbx,rdi          ; Guarda o ponteiro da estrutura "escondida".
    sub rsp,2072          ; Aloca espaço para a estrutura na pilha.
    mov rdi,rsp
    call memset

    lea r8d,[rbp+1]      ; r8d = x + 1;
    lea rdx,[rsp+4]       ; rdx aponta para o íncio de x[].
    lea rcx,[rsp+2052]    ; rcx aponta para o fim de x[].

.L1:
    mov [rdx],r8d         ; escreve em x[i].
    add rdx,4              ; avança i.
    cmp rdx,rcx            ; chegou ao fim?
    jne .L1                ; não? Continua preenchendo.

    mov rsi,rsp            ; rsi aponta para a estrutura na pilha.
    mov rdi,rbx            ; rdi aponta para estrutura "escondida".
    mov edx,2052           ; Vamos copiar 2052 bytes.
    mov [rsp],ebp           ; Escreve o parâmetro x em count.
    call memcp             ; Faz a cópia.
    add rsp,2072            ; Limpa a pilha.

    mov rax,rbx            ; Retorna o ponteiro para a estrutura "escondida".
    pop rbx
    pop rbp
    ret

```

Continuamos obtendo RAX como resposta, contendo o ponteiro “escondido” passado para a função, mas repare como ela ficou **tremendamente mais complicada do que deveria!** A função aloca 2052 bytes (os 512 *int's* do array e o *count*, da estrutura) mais 20 bytes (?!). Ela preenche todo o array com zeros (usando *memset*), coloca em R8D o valor de *x*, passado para a função e usa RDX e RCX nas iterações do loop, preenchendo o conteúdo da pilha com os valores desejados... Logo depois, *memcp* copia toda a estrutura para a região da memória dada pelo ponteiro “escondido” e retorna esse ponteiro em RAX.

E esse é um código simples! Às vezes o compilador se vê obrigado a fazer uso do seletor de segmento FS para conter um endereço base e torna todo o código praticamente ilegível, sem contar

pouco performático!

Agora, compare o código anterior com este:

```
struct myStruc *f(struct myStruc *ptr, int x)
{
    int i;

    ptr->count = x;
    for (i = 0; i < 512; i++)
        ptr->x[i] = x+1;
    return ptr;
}
-----%----- corte aqui -----%-----
f:
    mov rax,rdi
    mov [rdi],esi
    xor edx,edx
    lea ecx,[rsi+1]
.L1:
    mov [rax+4*rdx],ecx
    add rdx,4
    cmp rdx,2048
    jne .L1
    ret
```

Este código faz a mesma coisa (literalmente) que o anterior **sem** preencher o array *x* com zeros e **sem** fazer uma cópia para uma estrutura “escondida” nos parâmetros. Ela é menor e mais rápida!

Deixo essa dica: Jamais retorne estruturas e uniões por valor!

## O uso da pilha

A pilha é usada para receber parâmetros que não caibam num registradores ou se a quantidade de parâmetros for maior que uma certa quantidade, de acordo com a convenção de chamada. Repare que, para POSIX ABI, existem 5 registradores que suportam parâmetros inteiros e 8 registradores que suportam parâmetros em ponto flutuante. Se, por exemplo, tivéssemos uma função declarada como:

```
int f(int, int, int, int, int, float);
```

Os primeiros cinco parâmetros serão passados via RDI, RSI, RDX, R8 e R9, mas o sexto 'int' será, necessariamente, colocado na pilha. O último parâmetro será passado por XMM0.

Outro uso para a pilha é o armazenamento temporário e de variáveis locais (que não caibam nos registradores disponíveis para a rotina).

Quando o compilador tenta preservar valores contidos em registradores na pilha e recuperá-los depois, raramente usará instruções como PUSH e POP. O motivo é que essas instruções tomam, pelo menos, um ciclo de máquina adicional na manipulação do registrador RSP. Suponha que o compilador queira preservar RBX, RCX e RDX na pilha para recuperá-los depois. O código não otimizado padrão poderia ser mais ou menos assim:

```
push rbx    /* salva RBX, RCX e RDX na pilha. */
push rcx
push rdx
...
pop rdx    /* Recupera RBX, RCX e RDX da pilha. */
pop rcx
pop rbx
```

Cada uma das instruções PUSH subtrai 8 do registrador RSP e depois move o conteúdo do registrador sendo “empurrado”, usando RSP como ponteiro. O código equivalente do primeiro PUSH seria mais ou menos assim:

```
sub rsp,8
mov [rsp],rbx
```

No caso no código que precisa “empurrar” várias coisas para a pilha (e depois “puxá-las” de volta), o compilador geralmente resolve fazer a subtração (e depois a adição) uma só vez:

```
; Ajusta o RSP para 24 bytes 'para baixo'. Note que 24 é
; 3 vezes o tamanho de cada um dos registradores.
;
; O motivo de serem guardados 'de trás para frente' é que
; a pilha cresce 'para baixo'.

sub rsp,24
mov [rsp+16],rbx
mov [rsp+8],rcx
mov [rsp],rdx
...
mov rdx,[rsp]
mov rcx,[rsp+8]
mov rbx,[rsp+16]
add rsp,24
```

Em teoria, a primeira rotina, com os PUSHs e POPs, gastaria cerca de 12 ciclos, enquanto a rotina acima gastaria apenas 8.

## Variáveis locais e a pilha

Vimos como os argumentos de uma função são passados, seja por registradores, seja pela pilha, mas e quanto as variáveis locais?

Se você compilar seu código com algum nível de otimização, o compilador tentará manter variáveis locais dentro de registradores o tempo todo. Nem sempre ele consegue. Daí, a pilha também é usada para armazenar essas variáveis. A pilha é usada porque, quando a função termina, é feita a “limpeza” da pilha. Isso significa que as variáveis locais também estarão perdidas para a função chamadora.

Segundo as convenções de chamada, os argumentos são empilhados (de trás para frente, por default) seguido do endereço de retorno, quando for feito um *call*. Cada empilhamento é feito decrementando o registrador RSP e usando-o como ponteiro para armazenamento do valor empilhado... As variáveis locais são alocadas **depois** do endereço de retorno. Se [RSP+4] é a derreferência ao ponteiro para o primeiro argumento da função, na pilha, [RSP-4] é a derreferência à uma variável local, alocada na pilha. Considere o seguinte fragmento de código:

```
#include <stdio.h>

extern int g(void);

void f(void)
{
    int x = g();           // Chama g() para inicializar x.
    printf("%p\n", &x);   // Note que tomamos o endereço de x aqui.
}
-----%<---- corte aqui ----%<-----
.section .data

fmt:
    db    '%p',10,0

.section .text

extern g
extern printf
```

```

f:
    call g
    lea rsi,[rsp-4]      // Vamos passar o ponteiro para printf.
                           // Então, coloca-o em RSI (2º argumento de printf).
    mov [rsi],eax        // EAX contém o resultado retornado por g().
                           // Armazena EAX na variável local, na pilha.
    mov rdi,fmt           // RDI é o primeiro argumento de printf().
    call printf           // Chama printf().
    ret

```

No exemplo acima, o resultado da chamada à função *g()*, no registrador EAX, é armazenado na pilha, em [RSP-4], porque a função *printf* precisa do ponteiro da variável local *x*. Ainda, segundo a convenção de chamada, os dois argumentos de *printf* devem ser passados, pela ordem, em RDI e RSI. Por isso calculei RSI antes de usá-lo como ponteiro.

Como regra geral, toda variável local que seu código precisa acessar através de ponteiros é colocada na pilha...

Mesmo para aqueles casos onde o compilador consegue manter as variáveis locais em registradores, a região de armazenamento local (abaixo do endereço de retorno) pode ser usado como armazenamento temporário. Suponha que sua função use R12 e a função chamada também o use. É necessário guardar o conteúdo de R12 no espaço local, chamar a função e depois recuperá-lo:

```

mov [rsp-8],r12
call func
mov r12,[rsp-8]

```

A convenção de chamada usada pela arquitetura x86-64 prevê que alguns registradores podem ser descartados entre chamadas, mas outros devem ser preservados (lembra?). O artifício de usar a região de armazenamento local da pilha é um jeito de fazer isso. Outro jeito seria manter variáveis globais no segmento de dados, o que aumenta o tamanho do código e tem a possibilidade de poluir os caches<sup>32</sup>.

## O compilador não usa todas as instruções

Se você der uma olhada no volume 2 dos manuais de desenvolvimento de software da Intel para os modos i386 e x86-64 poderá encontrar instruções como XLAT, JECXZ, XCHG, LOOP e outras com nomes interessantes. Algumas parecem bem úteis, mas depois de olhar para muitas listagens em assembly geradas pelo seu compilador C, perceberá que ele não as usa... NUNCA! O motivo? Performance.

Algumas dessas instruções mais especializadas são realmente lentas. Mais lentas do que um código que faz a mesma coisa usando instruções mais “discretas”. Nos exemplos abaixo, a primeira rotina é mais rápida que a segunda:

```

; primeira rotina...
.L1:
; ... faz alguma coisa aqui ...
sub rcx,1
jnz .L1
-----%<---- corte aqui -----%<-----
; segunda rotina...
.L2:
; ... faz alguma coisa aqui ...
loop .L2

```

A instrução LOOP faz exatamente a mesma coisa que SUB/JNZ acima mas, surpreendentemente, é mais lenta!

---

<sup>32</sup> Sobre caches, veremos mais à frente.

A listagem abaixo mostra outro caso onde uma instrução especializada era mais lenta que a sequência de instruções mais discreta:

```
; primeira rotina...
bzero:
    mov rcx,rsi
    xor al,al
.L1:
    mov [rdi],al
    inc rdi
    dec rcx
    jnz .L1
    ret
-----%<---- corte aqui -----%<-----
; segunda rotina...
bzero:
    mov rcx,rsi
    xor al,al
    rep stosb
    ret
```

Como no caso anterior, a primeira rotina faz a mesma coisa que a segunda. Mas a primeira era mais rápida. Somente nas arquiteturas mais recentes (se não me engano, a partir da *Sandy Bridge*), o uso do prefixo REP com as instruções de manipulação de blocos STOSB e MOVSB ficaram mais rápidas...

A instrução XCHG é outro caso interessante... Aparentemente ela é bem útil: Troca o conteúdo entre dois registradores (ou registrador e memória). As duas listagens parciais, abaixo, seriam equivalentes:

```
...
; usando xchg para trocar RAX pelo conteúdo de memória
xchg rax,[var1]
...
-----%<---- corte aqui -----%<-----
...
mov rbx,[var1]
mov [var1],rax
mov rax,rbx
...
...
```

Exceto que a segunda listagem usa o registrador RBX como armazenamento temporário e a primeira não o faz. Nesse caso XCHG parece ser bem atraente, pois faz o que 3 "MOV" fazem, sem usar um registrador extra. Se fosse só isso ela seria, de fato, muito interessante!

Acontece que XCHG, quando um dos parâmetros é uma referência à memória, também modifica o estado do sinal LOCK# da CPU. Este sinal informa os outros processadores que o barramento de endereços e dados estão "travados" e não podem ser modificados. Isso garante que XCHG irá ler-modificar-gravar a memória sem interferências de outros processadores. Com isso, a instrução gasta mais ciclos de máquina do que deveria, ficando mais lenta que os 3 'MOV's.

```
xchg rax,[rdi]      ; Essa instrução usa o prefixo LOCK automaticamente.
xchg rdx,rcx         ; Essa instrução não usa o prefixo LOCK automático.
```

XCHG é muito boa quando temos que fazer trocas entre registradores. Estranhamente o compilador nunca a usa!

Existem ainda instruções mais complexas que não usadas pelo compilador: Seja pela baixa performance, pela "inutilidade" ou pelo uso muito específico (existem instruções que só funcionam no *ring 0*, por exemplo).

### ***Detalhes interessantes sobre a instrução NOP e o prefixo REP***

NOP é a instrução que não faz nada, só gasta tempo. Ela é bastante usada para alinhar código.

Tipicamente NOP é uma instrução que gasta apenas um byte no segmento de código, mas a Intel a estendeu para que possa usar mais espaço, se necessário, gastando a mesma quantidade de ciclos de clock.

Esses novos NOPs permitem operadores e são conhecidos como *hinted nops*. Eis exemplos da listagem obtida pelo NASM<sup>33</sup>:

```
1          bits 64
2          section .text
3
4 00000000 90      nop
5 00000001 660F1F00    nop word [rax]
6 00000005 660F1F0400    nop word [rax+rax]
7 0000000A 660F1F040500000000    nop word [nosplit rax+0]
8 00000013 0F1F00      nop dword [rax]
9 00000016 0F1F0400    nop dword [rax+rax]
10 0000001A 0F1F040500000000   nop dword [nosplit rax+0]
11 00000022 480F1F00    nop qword [rax]
12 00000026 480F1F0400    nop qword [rax+rax]
13 0000002B 480F1F040500000000   nop qword [nosplit rax+0]
```

Todas essas variações gastam apenas um único ciclo de clock, mas têm tamanhos diferentes. Repare que podemos usar NOPs de 1, 3, 4, 5, 8 e 9 bytes. Algumas dessas versões usam ou o prefixo 0x66 ou o prefixo 0x48. O prefixo 0x66 diz ao processador que operando terá 16 bits de tamanho.

O prefixo REX (entre 0x48 e 0x4f) é usado no modo de 64 bits para indicar que o operando tem, de fato, 64 bits de tamanho. O tamanho default de operandos, na arquitetura x86-64, é DWORD (32 bits). Por isso “NOP DWORD [RAX]” não tem prefixo REX, mas “NOP QWORD [RAX]” tem.

Não se preocupe com o prefixo REX ou o prefixo 0x66. Os compiladores tomam conta disso...

Para obter NOPs de 2 bytes podemos usar a instrução XCHG:

```
xchg ax,ax      ; Isso será traduzido para "66 90", no micro-código.
; É como se colocássemos o prefixo 66 antes do NOP.
```

Obter NOPs de 6 e 7 bytes de tamanho é mais complicado, mas é possível.

De fato, o NASM implementa alguns apelidos para esses tipos de NOPs: HINT\_NOPn, onde n pode ser um valor entre 1 e 63.

Outra instrução que pode ser usada para emular um NOP é LEA:

```
lea rax,[rax]
lea rax,[nosplit rax+0]
```

Embora essas instruções tendam a realizar uma operação aritmética (sem afetar quaisquer flags), o processador percebe que nada está sendo feito, de fato, e elas só gastarão 1 ciclo de máquina.

Quanto ao prefixo REP, às vezes, o compilador o coloca na frente de instruções onde ele não se aplica, como no exemplo:

```
rep ret
```

O que o compilador tenta fazer aqui também é alinhamento. Se ele usasse NOP para alinhar o RET, gastaria um ciclo de máquina adicional desnecessariamente... Ao usar REP, que só funciona com instruções de manipulação de bloco (MOVS, STOS, LODS, CMPS) e não afeta quaisquer outras, ele evita isso.

---

33 O modificador *nosplit* no endereço efetivo indica ao NASM que este não deverá “otimizar” a expressão.

## ***LOOP e LOOPNZ são diferentes, mas REP NZ e REP são a mesma coisa!***

E, por falar em manipulação de blocos, existem dois tipos de prefixo REP: Um que testa se o flag ZF é zero e outro se ele é um. Ao usar “REP” você está usando REP NZ... Mas, cuidado que com instruções que não afetam ZF, usa-se REP.

Usando as instruções de bloco com REP NZ ou REPZ o processador vai testar o conteúdo do registrador RCX contra zero, para determinar a quantidade máxima de repetições. Depois ele vai executar a instrução, decrementar RCX, incrementar/decrementar (dependendo do flag DF) RSI e/ou RDI (dependendo da instrução), testar o flag ZF e repetir tudo de novo, dependendo do prefixo. A instrução abaixo, pode ser escrita, em pseudo-código, como:

```
repnz scasb
-----%<---- corte aqui -----%<-----
/* pseudo-código */
extern struct flags eflags;

void repnz_scasb(unsigned long rcx, void *rdi, char al)
{
    while (rcx != 0)
    {
        rcx--;
        eflags.zf = 0;
        if (*rdi == al)
        {
            eflags.zf = 1;
            break;
        }
        if (!eflags.df)
            rdi++;
        else
            rdi--;
    }
}
```

LOOP, LOOPZ e LOOPNZ, por outro lado, são diferentes entre si. A primeira instrução testa somente se RCX é zero ou não, decrementando-o antes de saltar... As outras duas testam o conteúdo do flag ZF **antes** de decrementarem RCX.

## ***Assembly inline***

Dentro de seu código em C você pode usar códigos em assembly. Isso funciona na maioria dos compiladores C/C++, mas a sintaxe pode variar de compilador para compilador. Como estamos usando o GCC, a sintaxe pode parecer um tanto confusa:

```
_asm_ __volatile__ (
    "string_com_código_asm"
    : [saída]
    : [entrada]
    : [registadores modificados]
);
```

A palavra reservada `_asm_` é bastante óbvia, mas porque esse bloco deve ser marcado como `__volatile__`? Acontece que mesmo um código em assembly pode ser otimizado pelo GCC. Ao marcar o código como `volatile` dizemos ao compilador que este não deve realizar otimizações.

Mas, atenção! Marcar o bloco como `volatile` é apenas um pedido ao compilador que ele pode ignorar.

Outra coisa: Você pode usar as palavras reservadas “`asm`” e “`volatile`” sem os *underscores*:

```
asm volatile ( "string" : [saída] : [entrada] : [registros preservados] );
```

Só uso `_asm_` e `_volatile_` porque a documentação do GCC assim sugere que se faça, para evitar conflitos de *namespace*. Mas, se você der uma olhada no código fonte do kernel do Linux, por exemplo, verá essa última forma, visualmente mais confortável, sendo usada...

Em relação aos parâmetros contidos no bloco, a “string com código asm” é uma grande string contendo, obviamente, o código em assembly. A separação de linhas pode ser feita com um '\n' ou ';'. Particularmente, prefiro o último, mesmo que ele bagunce um pouco as listagens em assembly obtidas a partir do GCC... Só é necessário tomar cuidado com a plataforma onde esses códigos assembly inline serão compilados... O GAS (assembler usado para compilar o assembly inline) usa ';' como início de comentário em algumas plataformas. Assim, usar '\n' é mais seguro.

Toda função têm parâmetros de entrada, valores de retorno e recursos que precisam ser preservados. Um código em assembly inline não é diferente. Depois da string contendo o código em si, seguem três listas **opcionais**. Na primeira temos uma lista os valores de saída, na segunda os valores de entrada e na terceira a lista os registradores que devem ser preservados.

Cada item da listas de valores de entrada e de valores de saída contém uma string que descreve como o parâmetro deverá ser interpretado. Por exemplo:

```
unsigned int lo, hi, val;

...
__asm__ __volatile__ (
    "movl %2,%eax;"           /* lista de saídas */
    "movl %%rax,%%rdx;"       /* lista de entradas */
    "shrl 32,%%rdx"          /* lista de preservação. */
    : "=a" (lo), "=d" (hi)
    : "r" (val)
    : "rbx", "rcx"
);
}
```

Aqui temos a lista de saída que descreve que o valor em EAX (definido pela string “=a”) será colocado na variável 'lo' (porque 'lo' é definida como 'unsigned int') e o valor em EDX será colocado na variável 'hi' (veja o “=d”), também dependendo do tamanho de 'hi'.

Segue a lista de variáveis de entrada, que diz ao compilador que o conteúdo da variável 'val' será colocado em um registrador, à escolha do GCC (graças à string “r”), antes da execução do código em assembly.

E, finalmente, a lista de registradores que devem ser preservados. No caso, RBX e RCX. Um aviso é necessário quanto a essa lista de preservação... O GCC não sabe quais registradores serão modificados dentro da string do código no assembly inline, mas ele sabe que precisará usar os registradores e/ou memória definidos nos descritores de saída e entrada. Então, na lista de preservação precisamos listar apenas aqueles registradores que nosso código altera **que não estejam nas outras duas listas**.

Além de preservação de registradores, existem outros dois “registradores especiais” que podem ser usados nesse pedaço do bloco do assembly inline. Trata-se de “cc” e “memory”. O pseudo registrador “cc” é, na verdade, o registrador EFLAGS. Já “memory” é uma dica ao compilador quando o seu código faz alterações na memória que não são “previsíveis”, do ponto de vista do compilador... Por exemplo, se você usar instruções de bloco como SCAS ou MOVS, é prudente colocar “memory” na lista de preservação.

Os descritores de entrada e saída são meio estranhos, não? Os caracteres usados nessas strings variam de processador para processador, mas, de forma geral, para a arquitetura Intel (tanto i386 quanto x86-64) a tabela abaixo mostra os principais:

<b>Descriptor</b>	<b>Significado</b>
a, b, c, d, D e S	Respectivamente os registradores EAX, EBX, ECX, EDX, EDI e ESI ou seus derivados (de 16 ou 32 bits).
r	Um registrador qualquer, à escolha do compilador.
m	Uma referência à memória
g	Registrador, memória ou constante. O compilador escolhe.
i ou n	Valor inteiro (imediato).
f	Um “registrador” do x87, escolhido pelo compilador.
t	O “registrador” st(0) do x87.
u	O “registrador” st(1) do x87.
A	O par de registradores EDX:EAX ou DX:AX <sup>34</sup> .
x	Registrador SSE à escolha do compilador.
Yz	O registrador XMM0.

Tabela 6: Lista dos principais descritores de entra/saída para assembly inline

No caso da lista da descrição de saída, se usada, é necessário colocar um “=” ou um “+” antes do descritor. No caso de usar um “=”, dizemos ao compilador que o descritor será usado apenas para gravação (apenas como saída). O “+” diz que o descritor pode também ser lido pelo código em assembly. Normalmente usamos apenas “=”.

Descritores como 'r', 'g' e 'x' são interessantes porque permitem que você abstraia seu código do uso de um registrador específico, deixando o compilador escolher qual usar.

No exemplo dado anteriormente, o código assembly receberá o conteúdo da variável 'val' em um registrador qualquer, à escolha do compilador; ao terminar, o conteúdo de EAX será colocado em 'lo' e EDX em 'hi'; e os registradores RBX e RCX serão preservados (se necessário!). É simples assim.

O compilador sabe que deve usar EAX ao invés de RAX, AX, AH ou AL, por causa do tamanho da variável associada ao descritor. No exemplo a variável 'val' é do tipo *int* e, portanto, EAX é suficiente para comportá-la. Assim, os descritores “a”, “b”, “c”, “d”, “D” e “S” selecionam um dos registradores de uso geral de acordo com o tamanho e têm esses nomes genéricos.

Um meio de acessar, dentro do código assembly, um desses parâmetros descritos é através do número de ordem em que o descritor aparece em ambas as listas. Pense nesses descritores como se eles estivessem num array... No exemplo anterior o descritor associado com a variável 'lo' tem a posição 0 (zero), a posição 1 é a do descritor associado à variável 'hi' e o descritor de entrada, associado com a variável 'val' tem a posição 2.

Nosso código obtém o valor da variável 'val' assim:

```
movl %2,%eax;
```

Esse '%2' diz ao *assembler inline* que deve colocar, em seu lugar, o conteúdo do item 2 da lista de descritores. Note que o descritor é declarado como “r” e, portanto o compilador fará de tudo para substituí-lo por um registrador.

Esse macete é o responsável pelo fato de registradores terem que ser prefixados com dois ‘%’.

34 Note que 'A' **não** lida com o par RDX:RAX.

Além do uso do índice de um descritor dentro do código assembly, podemos usar esses índices nos próprios descritores. Por exemplo: Suponha que queiramos multiplicar um valor por 13, em assembly. Poderíamos fazer algo assim:

```
long val, result;
...
__asm__ __volatile__ (
    "cqo;"           /* Criação de um resultado temporário no ECX */
    "movl $13,%ecx;" /* Carrega o valor constante 13 para o ECX */
    "idiv %rcx"      /* Divide o valor de 'val' por 13 */
    : "=a" (result) /* A saída será copiada de RAX para 'result' */
    : "0" (val)      /* A entrada será copiada de 'val' para RAX */
    : "rcx"          /* Informa que RCX foi alterado na rotina também!
);

```

Eu disse que os descritores são strings, certo? E strings podem ter mais que um caracter... É possível refinarmos o descritor combinando as descrições acima com outras:

Descritores adicionais	Significado
I	Inteiro na faixa de 0..31.
J	Inteiro na faixa de 0..63
K	Inteiro na faixa de -128..127 (8 bits)
N	Inteiro na faixa de 0..255 (8 bits)
G	Constante de 80 bits (x87)
C	Constante (ponto flutuante) para ser usada em instrução SSE
e	Constante de 32 bits sinalizada.
Z	Constante de 32 bits sem sinal.

Tabela 7: Descritores adicionais que podem ser combinados com outros.

Assim, poderíamos usar um descritor “dN” como entrada que um valor pode ser compilado como o registrador DL (ou DH) ou um valor inteiro. É o caso de:

```
__asm__ __volatile__ ( "outb %%al,%1" : : "a" (value), "dN" (port) );
```

Se fizermos “port” ser 0x70, por exemplo, o compilador pode escolher compilar a instrução acima como “OUT 0x70,AL” ao invés de “OUT DX,AL”. Então, fique atento: Os descritores não somente informam entradas e saídas, mas têm o potencial de modificar o próprio código em assembly...

## Operador de indireção (ponteiros) em assembly

Assim como em C, assembly também possui uma sintaxe particular para lidar com ponteiros. Vale lembrar dois detalhes: Primeiro, “ponteiro” é só um outro nome para “endereço” e, em segundo lugar, o operador de indexação de arrays ‘[]’, em C, é apenas uma notação diferente para lidar também com ponteiros. Com assembly é a mesma coisa. O operador de indireção do assembly é ‘[]’.

Isso é particularmente importante quando usamos o NASM. As duas declarações abaixo, que fazem referência à mesma “variável”, têm significados diferentes:

```
mov rax,var1 ; RAX receberá o endereço de 'var1'.
mov rax,[var1] ; RAX receberá o valor contido no endereço de 'var1'.
```

Essa diferença fica evidente quando usamos um registrador como ponteiro. Suponha que RDI tenha o endereço de uma variável. Para acessar o conteúdo desse endereço temos que usar o operador de indireção ‘[]’ cercando o nome do registrador:

```
mov rax,[rdi] ; RAX receberá o 'long' cujo endereço é dado por RDI.
```

## Usando a sintaxe Intel no assembly inline do GCC

Por default o GCC suporta o estilo AT&T da linguagem assembly. Neste estilo, a ordem dos operadores numa instrução é invertida. Ainda, a instrução em si carrega um modificador dizendo o tamanho dos operandos. Por exemplo:

```
movl $4,%eax ; é a mesma coisa que 'mov eax,4'.
```

O 'l', depois de 'mov', significa *long*. O valor 4, precedido de um '\$', é uma constante que será colocada no EAX.

Outra chatice do estilo AT&T é a notação usada para referencias à memória. Ao invés de usar o operador de indireção '['] e a notação Intel:

```
[BASE + ÍNDICE*MULTIPLICADOR + DESLOCAMENTO]
```

Que parece-me ser mais intuitiva que o padrão usado no sabor AT&T, que usa notação diferente. Algo assim:

```
DESLOCAMENTO(BASE, ÍNDICE, MULTIPLICADOR)
```

Veja a diferença nas duas instruções, idênticas:

```
mov rax,[rbx + rsi*2 + 3] ; padrão Intel.  
movq 3(%rbx,%rsi,2),%rax ; padrão AT&T
```

A segunda instrução requer mais atenção para ser entendida do que a primeira, não? Suspeito que a sintaxe AT&T deriva do padrão do assembly dos antigos processadores MC68000.

Para usar o estilo Intel, parecido com o estilo usado no NASM e em outros assemblers, no *assembler inline* do GCC, basta colocar seu código entre as diretivas ".intel\_syntax" e ".att\_syntax". Eis um exemplo simples que copia uma variável para outra:

```
unsigned v1=3, v2;  
  
/* Estilo AT&T (default) */  
__asm__ __volatile__ (  
    "movl %1,%eax;"  
    "movl %%eax,%0;"  
    : "=g" (v2) : "g" (v1) : "rax"  
);  
  
/* Estilo Intel */  
__asm__ __volatile__ (  
    ".intel_syntax;"  
    "mov eax,%1;"  
    "mov %0,eax;"  
    ".att_syntax;"  
    : "=g" (v2) : "g" (v1) : "rax"  
);
```

É necessário dizer ao assembler inline que o estilo AT&T deve ser retomado, senão você obterá erros de compilação depois do bloco `__asm__`.

## Problemas com o assembler inline do GCC

Lembre-se que o GCC tende a otimizar todo o código que ele traduz, **inclusive** o código contido no bloco `__asm__`. Tentamos evitar isso com o uso de `__volatile__`, mas nem sempre o compilador nos obedece. Isso pode ser bastante problemático para aqueles que gastaram horas refinando uma rotina para tentar obter a menor quantidade de ciclos de máquina possível e verem seu trabalho jogado no

lixo pelo compilador (ou, pior, nem perceber que o compilador te sacaneou!).

Repare que o próprio esquema de passagem de parâmetros para o bloco `__asm__` implica que alguma otimização será feita, quer você queira, quer não.

Todo o sentido de ter um assembler inline é que você pode colocar seus códigos já otimizados, em assembly, junto com seu código C, observando as regras do compilador... Se o compilador vai otimizar seu código de qualquer maneira, então, pra que usar assembly?

Recomendo que mantenha seus códigos com a menor quantidade possível de assembly inline. Se quiser colocar uma rotina em assembly no seu programa, use um assembler externo como o NASM. Reserve o assembler inline para aquelas rotinas que fazem algo que o compilador não faria com facilidade. Um exemplo é o código fonte do kernel do Linux: Assembly inline só é usado lá para códigos que o compilador normalmente não gera. Por exemplo, usando instruções que só estão disponíveis no nível mais privilegiado.

## Usando NASM

Ao invés de usar o assembler inline, criar nossas rotinas em assembly em um código fonte separado, oferece também suas vantagens. Em primeiro lugar, não há dependência dos recursos de otimização do GCC (ou de seu compilador C/C++ preferido). Outra coisa interessante é que o NASM é *cross platform*. Isso quer dizer que podemos criar códigos em assembly para serem usados no Linux, OS/X, Windows e outros ambientes com pouca ou nenhuma modificação no código original<sup>35</sup>.

A preferência pelo NASM é que ele é um desses compiladores “free software” que estão disponíveis em qualquer plataforma (assim como o GCC) e oferece uma sintaxe parecida, com diferenças interessantes, em relação ao MASM (Macro Assembler). Existem outros “sabores” de assemblers, como o YASM e o FASM, bem como o MASM e TASM (esses últimos, disponíveis apenas para Windows!). Você pode preferir usar o GAS (GNU Assembler), mas prepare-se para a confusão da sintaxe AT&T. Prefiro uma sintaxe mais parecida possível com a da Intel.

Um código escrito para o NASM é bem simples. Existem diretivas para controle do compilador e do código em si. Um programa típico, para o NASM é mais ou menos assim:

```
; triple.asm
bits 64          ; Vamos usar x86-64.
section .text    ; Aqui começa o 'segmento' de código.

global f:function ; O label 'f' será "exportado" como ponto de entrada de uma função.

; A função abaixo equivale a:
; int f(int x) { return 3*x; }

f:
    mov eax,edi
    add eax,eax
    add eax,edi
    ret
```

Basicamente existem quatro tipos de “sections” ou “segmentos”: Código, dados inicializados, dados não inicializados e constantes. Estes são especificados na diretiva SECTION (ou SEGMENT) como '.text', '.data', '.bss' ou '.rodata', respectivamente. A sigla BSS é originária de *Block Started by Symbol*. Isso quer dizer que o segmento `.bss` **não** faz parte da imagem binária do seu programa contido no arquivo em disco... Esse segmento é criado e preenchido com zeros pela biblioteca `libc`... Dados “não inicializados”, em C, são sempre inicializados com zeros!

Diferente de outros assemblers, não existem diretivas específicas para demarcar blocos de *procedimentos* ou funções além da declaração da SECTION `.text`. No MASM (Macro Assembler) a

<sup>35</sup> Isso não é lá muito verdadeiro. Sistemas operacionais têm esquemas diferentes. Por exemplo, como já vimos, na convenção de chamada!

listagem acima seria escrita mais ou menos assim (para i386):

```
.586p
.model flat,stdcall
.code
; Exporta o procedimento 'f'.
public f
; No MASM é necessário usar PROC e ENDP para demarcar uma função.
f proc near
    mov eax,edi      ; RDI é o primeiro parâmetro também na convenção da Microsoft!
    add eax,eax
    add eax,edi
    ret
f endp
end
```

Note os “proc near” e “endp”...

No caso do NASM um ponto de entrada de função é simplesmente um símbolo como qualquer outro. A única coisa que você precisa fazer é declarar o símbolo como “global” se for usá-lo em outro lugar e, opcionalmente, dizer que este símbolo está relacionado a uma função (o atributo “:function”, depois do nome da função na diretiva “global”, no exemplo). Daí, para usar a função no seu código em C, basta declarar o protótipo como 'extern' e depois linkar tudo. Eis um exemplo:

```
# Makefile
test: test.o triple.o
    gcc -o $@ $^

test.o: test.c
    gcc -O3 -march=native -c -o $@ $<

triple.o: triple.asm
    nasm -f elf64 -o $@ $<
-----%<---- corte aqui -----%<-----
/* test.c */
#include <stdio.h>

/* Função definida em triple.asm, lá em cima. */
extern int f(int);

int main(int argc, char *argv[])
{
    printf("3*23 = %d\n", f(23));
    return 0;
}
```

O compilador vai criar um arquivo objeto contendo a imagem binária do seu código fonte e os símbolos que foram “publicados”. No caso do Linux o formato do arquivo objeto tem que ser ELF. No caso do Windows, o formato é WIN64<sup>36</sup>. Os formatos default do NASM são ELF32 e WIN32, para Linux e Windows, respectivamente. Por isso você terá que usar a opção '-f' do assembler para gerar o arquivo objeto compatível com o modo de 64 bits.

Podemos ter problemas com o formato escolhido... Os formatos WIN32 ou WIN64, por exemplo, não suportam atributos nos símbolos exportados. Se você tentar compilar o *triple.asm*, como está, no formato WIN64, e obterá:

```
$ nasm -f win64 triple.asm -o triple.o
triple.asm:9: error: COFF format does not support any special symbol types
```

Para evitar isso você pode retirar o atributo 'function' na diretiva 'global'. Esses atributos existem

<sup>36</sup> Linux usa o padrão *Executable and Linkable Format* (ELF). Windows usa o *Common Object File Format* (COFF). O formato COFF foi definido pelo POSIX e adotado pela Microsoft com o surgimento do Win32. No Linux o formato ELF foi elaborado como um avanço sobre o COFF. Outros sistemas UNIX podem usar COFF também.

para fins de otimização e documentação feitas pelo linker, especialmente para o formato ELF. Aparentemente não têm quaisquer usos no formato COFF.

# Capítulo 6: Ferramentas

Eis algumas ferramentas úteis para o desenvolvedor, incluindo debugger, profiler, criação de projetos e análise de código.

## GNU Linker (*ld*)

É comum usarmos o GCC para compilar e, depois, linkar os diversos módulos num único arquivo executável ou biblioteca. Esse é um atalho conveniente para o desenvolvedor, no entanto, o que ocorre por debaixo dos panos é uma chamada ao linker (*ld*).

O linker faz algumas coisas: Ele sabe quais segmentos (ou “sessões”) estão contidas no arquivo alvo (executável ou biblioteca) e sabe como esses segmentos devem ser organizados. Ele também mantém alguns símbolos especiais (*\_start*, *\_end*, etc) onde, por exemplo, o símbolo *\_start* é o ponto de entrada default para execução. Outra tarefa do linker é misturar os blocos de dados e código contidos nas sessões dos arquivos objeto parciais, bem como resolver símbolos “extern” de um arquivo objeto, fazendo referência na definição desse símbolo definido em outro arquivo objeto (ou mostrar um erro se não encontrar a definição do símbolo em lugar algum!).

Mas, onde essas informações são definidas?

O linker precisa de um *script* que contenha essas definições. Se você compilar um pequeno programa exemplo dizendo ao linker par mostrar o que ele está fazendo (-Wl,--verbose), verá uma série de informações do linker, incluindo os tipos de “alvos” suportados e o script *default* usado. Esse script é grande e não vale a pena ser reproduzido aqui. Ao invés disso, eis um script do linker customizado para o código de boot (16 bits) do kernel do Linux:

```
/*
 * setup.ld
 *
 * Linker script for the i386 setup code
 */
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)

SECTIONS
{
    . = 0;
    .bstext      : { *(.bstext) }
    .bsdata       : { *(.bsdata) }

    . = 495;
    .header      : { *(.header) }
    .entrytext   : { *(.entrytext) }
    .inittext    : { *(.inittext) }
    .initdata    : { *(.initdata) }
    __end_init = .;

    .text         : { *(.text) }
    .text32       : { *(.text32) }

    . = ALIGN(16);
    .rodata      : { *(.rodata*) }
    .videocards  :
        video_cards = .;
        *(.videocards)
        video_cards_end = .;
}

    . = ALIGN(16);
    .data        : { *(.data*) }
```

```

.signature      : {
    setup_sig = .;
    LONG(0x5a5aaa55)
}

.= ALIGN(16);
.bss           :
{
    __bss_start = .;
    *(.bss)
    __bss_end = .;
}

.= ALIGN(16);
.end = .;

/DISCARD/ : { *(.note*) }

/*
 * The ASSERT() sink to . is intentional, for binutils 2.14 compatibility:
 */
.= ASSERT(_end <= 0x8000, "Setup too big!");
.= ASSERT(hdr == 0x1f1, "The setup header has the wrong offset!");
/* Necessary for the very-old-loader check to work... */
.= ASSERT(_end_init <= 5*512, "init sections too big!");
}

```

Esse script define um conjunto de segmentos (sessões) que estarão contidas no arquivo “alvo” (*vmlinuz?!*), a ordem em eles aparecerão no arquivo depois dos códigos linkados e um conjunto de símbolos. Os segmentos são *.bstext*, *.bsdata*, *.header*, *.entrytext*, *.inittext*, *.initdata*, *.text*, *.text32*, *.rodata*, *.videocards*, *.data*, *.signature*, e *.bss*. Os segmentos *.bstext* e *.bsdata* começam no endereço virtual 0 (zero). O ‘.’, no script, é um contador de endereços. O segmento *.header* começa no endereço virtual 495 (0x1EF) e os demais segmentos o seguem...

Os símbolos definidos no script são *\_start* (a diretiva ENTRY diz que esse é o ponto de entrada do programa linkado), *\_end\_init* (que será o endereço do final do segmento de dados *.initdata*), *\_bss\_start* e *\_bss\_end* (que são definidos para que os códigos de inicialização tenham a chance de zerar todo o segmento) e *\_end* que é o endereço final da imagem binária<sup>37</sup>.

Embora o script acima seja customizado para o kernel, no *userspace* o linker tem um script default onde um monte de segmentos e símbolos definidos para arquivos executáveis e bibliotecas. Por exemplo, para executáveis o linker usa um script que define o endereço virtual de carga inicial em 0x400000. No caso de *shared libraries* o endereço inicial não é definido, já que é necessário que esse tipo de arquivo contenha código *independente de posição* (PIC = *Position Independent Code*).

Para ver o script que o linker está usando, chame-o com a opção *-verbose*, na hora de linkar... Se você usa o próprio GCC para linkar objetos, use a opção “*-Wl,--verbose*”.

Para executáveis, já que os símbolos *\_start* e *\_end* são definidos pelo script default do linker<sup>38</sup>, podemos acessá-los em nossos programas facilmente:

```

/* Mostra endereços de início e fim dos segmentos do nosso programa */
#include <stdio.h>

/* O “tipo” dos símbolos é irrelevante. Para obter o endereço do símbolo
é sempre necessário usar o operador &. Poderíamos declará-los como
“void *”, por exemplo. */
extern unsigned long _start; /* Ponto de entrada */
extern unsigned long __executable_start, _end; /* Início e fim dos segmentos. */

```

---

<sup>37</sup> De posse de símbolos como *\_\_executable\_start* e *\_end* você poderia calcular o tamanho total da memória em uso, mas não deixe de considerar que sua aplicação pode fazer uso de alocação dinâmica!

<sup>38</sup> Obviamente, os símbolos *\_start* e *\_\_executable\_start* não são definidos para bibliotecas.

```

void main(void)
{
    printf("Endereço do início: %p\n"
           "Endereço do fim: %p\n"
           "Ponto de entrada: %p\n"
           "Endereço de main(): %p\n",
           &__executable_start, &_end, &_start, main);
}
-----%<---- corte aqui -----%>-----
$ gcc -o test test.c
$ ./test
Endereço do início: 0x400000
Endereço do fim: 0x600960
Ponto de entrada: 0x40042b
Endereço de main(): 0x400506

```

Para obter o endereço linear do símbolo é necessário usar o operador `&`, como você pode ver...

Se você quiser ver o conjunto de símbolos definidos em seu executável (ou num arquivo objeto) pode usar o utilitário *objdump* com a opção `-t` (desde que os símbolos não tenham sido extirpados do executável final com a opção `-s` do linker).

## **Obtendo informações com objdump**

O utilitário *objdump* pode listar uma série de informações interessantes sobre arquivos objeto, incluindo executáveis e bibliotecas. Lista de segmentos (opção `-h`), lista de símbolos (opção `-t`) e até mesmo o disassembly dos códigos contidos no arquivo (opções `-d`, `-D` ou `-S`). A opção `-d` faz o disassembly apenas das sessões “executáveis” (isso é definido no script do linker!), já a opção `-D` tenta “disassembler” **todas** as sessões. A opção `-S` é usada caso o seu programa tenha informações de debugging embutidas: Além do disassembly essa opção listará o código fonte correspondente também.

No caso do disassembly, podemos usar uma opção `-M` para dizer o “sabor” que queremos usar. O “sabor” default é o formato AT&T. Para o sabor “Intel” use “`-M intel`”. Note também que as opções `-S` e `-d` listarão **todas** as sessões “executáveis” e **todos** os símbolos contidos no arquivo.

## **GNU Debugger**

Seus códigos quase nunca se comportam como você espera. Por mais cuidado que se tenha, bugs são quase sempre incluídos naquelas rotinas que você pensa serem perfeitas. Até o momento que você compila e executa o programa e percebe que ele não está fazendo o que deveria.

Para consertar o código podemos usar um debugger (literalmente, um “tirador de bugs”). Um dos mais poderosos e *free* é o GNU Debugger (GDB).

Usar o GDB exige alguma preparação: A primeira coisa que você terá que fazer é inserir, no seu código executável, as *informações de debugging*. Isso é feito usando a opção `'-g'` durante a compilação. É útil também não usar as otimizações automáticas do compilador. Essas otimizações podem extirpar parte do código que você quer debugar, tornando a sessão de *debugging* mais complicada. Lembre-se que, nessa etapa, o que queremos fazer é retirar erros de nosso programa... Ainda não queremos fazê-lo ficar rápido ou eficiente. Assim, usar a opção `'-O0'` também é útil (já que, por default, o compilador usa `'-O1'`).

Não mostrarei aqui técnicas de debugging. Existem publicações mais interessantes do que eu poderia escrever em poucos tópicos. O que nos interessa é usar o GDB como disassembler e alguns comandos para obter informações do código.

## Configurando o GDB

Prefiro a sintaxe *Intel* ao invés da sintaxe AT&T nas listagens de códigos em assembly. Para configurar o GDB, basta usar o comando:

```
set disassembly-flavor intel
```

O chato é que toda vez que for usar o *gdb* você deveria digitar essa linha. Felizmente, existe uma maneira de executar certos comandos automaticamente sempre que o *gdb* for iniciado. Basta colocá-los no arquivo de configuração *~/.gdbinit*:

```
$ echo 'set disassembly-flavor intel' > ~/.gdbinit39
```

## Duas maneiras de listar código assembly no gdb

Suponha que você queira listar o famoso código “hello, world”:

```
/* test.c */
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello, world.\n");
    return 0;
}
-----%<---- corte aqui ----%<-----
$ gcc -g -o test.c -o test
```

Uma vez compilado com a opção *-g* do *gcc*, você pode carregar o executável e listá-lo usando o comando *disassemble* (ou a versão abreviada: “*disas*”):

```
$ gdb ./test
...
(gdb) dasas main
Dump of assembler code for function main:
0x0000000000400400 <+0>: sub    rsp,0x8
0x0000000000400404 <+4>: mov    edi,0x4005cc
0x0000000000400409 <+9>: call   0x4003e0 <puts@plt>
0x000000000040040e <+14>: xor    eax,eax
0x0000000000400410 <+16>: add    rsp,0x8
0x0000000000400414 <+20>: ret
End of assembler dump.
```

GDB fornecerá uma listagem contendo o endereço das instruções, o offset relativo a partir do início da função (porque, no modo x86-64, é possível usar um modo de endereçamento relativo ao registrador RIP) e as instruções. Acontece que GDB permite listar mais detalhes! Se quisermos listar o código fonte, em C, que gerou a listagem acima, basta usarmos a opção */m*:

```
(gdb) dasas /m main
Dump of assembler code for function main:
4
{
    0x0000000000400400 <+0>: sub    rsp,0x8
    5      printf("Hello\n");
    0x0000000000400404 <+4>: mov    edi,0x4005cc
    0x0000000000400409 <+9>: call   0x4003e0 <puts@plt>
    6      return 0;
    7
    0x000000000040040e <+14>: xor    eax,eax
    0x0000000000400410 <+16>: add    rsp,0x8
    0x0000000000400414 <+20>: ret
```

Também podemos acrescentar o conteúdo binário das instruções usando a opção */r*:

---

<sup>39</sup> Repare que o arquivo *~/.gdbinit* pode já existir em seu sistema. Se este for o caso, mofidique-o com seu editor de textos favorito.

```
(gdb) disas /r main
Dump of assembler code for function main:
0x0000000000400400 <+0>: 48 83 ec 08    sub    rsp,0x8
0x0000000000400404 <+4>: bf cc 05 40 00 mov    edi,0x4005cc
0x0000000000400409 <+9>: e8 d2 ff ff ff call   0x4003e0 <puts@plt>
0x000000000040040e <+14>: 31 c0           xor    eax, eax
0x0000000000400410 <+16>: 48 83 c4 08    add    rsp,0x8
0x0000000000400414 <+20>: c3               ret
```

## Listando registradores

Uma vez que sua sessão de debugging está em andamento, você pode listar os registradores com o comando “info”. Existem duas maneiras: “info reg” e “info all-reg”. Nem todos os registradores são listados. No primeiro caso você obterá os registradores de uso geral, RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8 até R15, RIP e “EFLAGS” (não RFLAGS! Mas não importa, já que este é, essencialmente EFLAGS com os 32 bits superiores zerados!). No segundo caso, os registradores de ponto flutuante e SIMD (SSE ou AVX, dependendo de sua arquitetura) são listados também:

```
#include <stdio.h>

int f(int x)
{
    return 2*x;
}

int main(int argc, char *argv[])
{
    printf("%d\n", f(2));

    return 0;
}
```

Compilando e iniciando nossa sessão de debugging:

```
$ gcc -O1 -g test.c -o test
$ gdb test
Reading symbols from test...done.
(gdb) b f
Breakpoint 1 at 0x40050c: file test.c, line 5.
(gdb) r
Starting program: test
Breakpoint 1, f (x=2) at test.c:5
5      return 2*x;

(gdb) info reg
rax            0x7fffff7dd9ec8 140737351884488
rbx            0x0              0
rcx            0x0              0
rdx            0x7fffffff1c8 140737488347592
rsi            0x7fffffff1b8 140737488347576
rdi            0x2              2
rbp            0x7fffffff0b0 0x7fffffff0b0
rsp            0x7fffffff0b0 0x7fffffff0b0
r8             0x7fffff7dd8300 140737351877376
r9             0x7fffff7deb310 140737351955216
r10            0x0              0
r11            0x7fffff7a6fdb0 140737348304304
r12            0x400400          4195328
r13            0x7fffffff1b0 140737488347568
r14            0x0              0
r15            0x0              0
rip            0x400513          0x400513 <f+7>
eflags         0x206            [ PF IF ]
cs             0x33             51
ss             0x2b             43
ds             0x0              0
es             0x0              0
fs             0x0              0
gs             0x0              0
```

```
(gdb) disas f
Dump of assembler code for function f:
=> 0x00000000040050c <+0>: lea    eax,[rdi+rdi*1]
    0x00000000040050f <+3>: ret
End of assembler dump.
```

Como era esperado, graças à convenção de chamada, o registrador RDI contém o primeiro parâmetro da chamada para a função f() - marcado em vermelho para chamar sua atenção. Repare que no disassembler da função o breakpoint está marcado com “=>”.

Outra maneira de inspecionar o conteúdo de um registrador é usando o comando *print* e o nome do registrador precedido com \$, como \$rax.

## **Examinando a memória com o GDB**

Além do disassembler e da possibilidade de examinar o conteúdo dos registradores, o comando “x” do GDB nos permite examinar uma região da memória. Basta fornecer o formato, a quantidade e o endereço (ou o símbolo) desejado. Com o mesmo código acima, poderíamos ver o conteúdo da pilha quando o breakpoint é alcançado:

```
$ gdb test
Reading symbols from test...done.
(gdb) b f
Breakpoint 1 at 0x40050c: file test.c, line 5.
(gdb) r
Starting program: test
Breakpoint 1, f (x=2) at test.c:5
5      return 2*x;
(gdb) x/32bx $rsp
0x7fffffffddde0: 0x00 0xde 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffddde8: 0x54 0x05 0x40 0x00 0x00 0x00 0x00 0x00
0x7fffffffddf0: 0xe8 0xde 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffddf8: 0x00 0x00 0x00 0x00 0x01 0x00 0x00 0x00
```

Usei o registrador RSP como endereço base para o exame, pedindo para examinar 32 bytes e mostrá-los em hexadecimal (x/32bx). Podemos usar as letras 'b', 'w', 'h' e 'q' para BYTE, WORD, DWORD (a letra 'd' é usada para “decimal”) e QWORD, respectivamente. O 'x' final diz que os valores devem ser impressos em hexadecimal.

Podemos usar um tamanho e a letra 's' para mostrar o bloco em formato de string. As letras 'd' e 'u' mostram os dados em decimal com e sem sinal, respectivamente. A letra 't' mostra os valores em binário (a letra é devido a palavra TWO). A outra formatação interessante é 'f', para ponto flutuante (embora, para mim, não esteja claro qual é o formato real: *float*, *double* ou *long double*?).

## **Obtendo listagens em assembly usando o GCC**

Um recurso muito útil para análise do seu próprio código é pedir ao compilador que gere uma listagem em assembly. No GCC você pode fazer isso usando a opção '-S'. No Visual C++ basta usar a opção '-Fa' com o CL.EXE.

Minha preferência, no caso do GCC, é gerar listagens no estilo Intel, ao invés do estilo AT&T. Isso pode ser obtido adicionando a opção '-masm=intel'. E o GCC ainda permite acrescentar comentários à listagem gerada usando-se a opção '-fverbose-asm'. Outra coisa importante é dizer ao compilador qual arquitetura será usada para otimização. Isso é feito através da opção '-march'. Eu prefiro usar “-march=native” para que o GCC use, como processador alvo, o da máquina em que o código foi compilado. Se seu processador suporta a extensão AVX ou AVX-512, por exemplo, o GCC usará as instruções estendidas.

Num executável de produção você pode querer escolher um processador ou arquitetura específica. Consulte a documentação do GCC para saber quais estão disponíveis.

Eis um exemplo de geração de listagem assembly, usando o GCC, para o seguinte código:

```
/* list.c */
#include <malloc.h>

/* A estrutura de nosso 'nó'. */
struct node_s {
    struct node_s *next;
    void *data;
    size_t size;
};

/* Insere um nó depois do nó dado por node_ptr. */
struct node_s *InsertNodeAfter(struct node_s *node_ptr, void *data, size_t size)
{
    struct node_s *newnode;

    if ((newnode = (struct node_s *)malloc(sizeof(struct node_s))) != NULL)
    {
        newnode->next = node_ptr->next;
        newnode->data = data;
        newnode->size = size;
        node_ptr->next = newnode;
    }

    return newnode;
}
-----%<---- corte aqui -----%<-----
$ gcc -O3 -march=native -S -masm=intel -fverbose-asm list.c
```

A listagem abaixo, compilada para um processador i7-4770, fica mais ou menos assim (aliás, para todos os modelos de processadores que suportam x86-64):

```
; Entrada: RDI = node_ptr,
;           RSI = data
;           RDX = size
; Retorna RAX.
InsertNodeAfter:
; Salva os parâmetros em R12, RBP e RBX porque esses registradores têm que
; ser preservados entre chamadas de funções. A função malloc(), abaixo, pode
; modificar RDI, RSI e RDX!
push r12
mov r12,rsi      # data, data
push rbp
mov rbp,rdx      # size, size
push rbx
mov rbx,rdi      # node_ptr, node_ptr

; Aloca 24 bytes (sizeof(struct node_s)).
mov edi,24
call malloc

test rax,rax      # new_node
je .insert_exit ; Se RAX == NULL, sai.

mov rdx,[rbx]      # D.2521, node_ptr_5(D)->next
mov [rax+8],r12    # newnode_4->data, data
mov [rax+16],rbp   # newnode_4->size, size
mov [rax],rdx     # newnode_4->next, D.2521
mov [rbx],rax     # node_ptr_5(D)->next, newnode

.insert_exit:
pop rbx
pop rbp
pop r12
ret
```

Os comentários mais claros são colocados pelo GCC e os **em vermelho** são meus. Note que o significado de uma instrução ou referência à memória é comentada com o nome da variável, do código original em C, referenciada ou um nome esquisito (é o caso de D.2521, acima) usado para uma variável temporária.

Outra maneira é pedir ao compilador que use o *GNU Assembler* (GAS) para gerar uma listagem

mista contendo o seu código fonte como comentários. Isso é feito adicionando as opções `-g` e `-Wa,-ahlnd`, no gcc, e redirecionando a saída para um arquivo texto. Eis o mesmo exemplo:

```
$ gcc -g -O3 -march=native -masm=intel -Wa,-ahlnd -c list.c > list.lst
```

É necessário o uso da opção `-g` para que o *GNU Assembler* tenha como mesclar o código fonte, em C, com a listagem em assembly. A linha de comando acima criará algo assim:

```
$ cat list.lst
1                      .file "list.c"
2                      .intel_syntax noprefix
3                      .text
4                      .Ltext0:
5                          .p2align 4,,15
6                          .globl InsertNodeAfter
7                      InsertNodeAfter:
8                      .LFB24:
9                          .file 1 "ins.c"
10                     ****/* insafter.c */
11                     ****#include <malloc.h>
12                     ****
13                     **** /* A estrutura de nosso 'nó'. */
14                     **** struct node_s {
15                         struct node_s *next;
16                         void *data;
17                         size_t size;
18                     };
19                     ****
20                     **** /* Insere um nó depois do nó dado por node_ptr. */
21                     **** struct node_s *InsertNodeAfter(struct node_s *node_ptr, void *data,
22                                         size_t size)
23                     **** {
24                         .loc 1 13 0
25                         .cfi_startproc
26                         .LVL0:
27                             push r12
28                             .cfi_def_cfa_offset 16
29                             .cfi_offset 12, -16
30                             mov r12, rsi
31                             push rbp
32                             .cfi_def_cfa_offset 24
33                             .cfi_offset 6, -24
34                             mov rbp, rdx
35                             push rbx
36                             .cfi_def_cfa_offset 32
37                             .cfi_offset 3, -32
38                             .loc 1 13 0
39                             mov rbx, rdi
40                             **** struct node_s *newnode;
41                             ****
42                             if ((newnode =
43                                 (struct node_s *)malloc(sizeof(struct node_s))) != NULL)
44                             .loc 1 16 0
45                             mov edi, 24
46                             .LVL1:
47                             call malloc
48                             .LVL2:
49                             test rax, rax
50                             je .L6
51                             **** {
52                             newnode->next = node_ptr->next;
53                             .loc 1 18 0
54                             mov rdx, QWORD PTR [rbx]
55                             newnode->data = data;
56                             .loc 1 19 0
57                             mov QWORD PTR [rax+8], r12
58                             newnode->size = size;
59                             .loc 1 20 0
60                             mov QWORD PTR [rax+16], rbp
61                             newnode->data = data;
62                             .loc 1 18 0
63                             mov QWORD PTR [rax], rdx
```

```

21:ins.c      ****      node_ptr->next = newnode;
42                  .loc 1 21 0
43 002a 488903      mov     QWORD PTR [rbx], rax
44      .L6:
22:ins.c      ****  }
23:ins.c      ****
24:ins.c      ****  return newnode;
25:ins.c      ****  }
45                  .loc 1 25 0
46 002d 5B      pop    rbx
47                  .cfi_def_cfa_offset 24
48      .LVL3:
49 002e 5D      pop    rbp
50                  .cfi_def_cfa_offset 16
51      .LVL4:
52 002f 415C      pop    r12
53                  .cfi_def_cfa_offset 8
54      .LVL5:
55 0031 C3      ret
56                  .cfi_endproc
57      .LFE24:
59      .Ltext0:
60      .file 2 "/usr/lib/gcc/x86_64-linux-gnu/4.8/include/stddef.h"
61      .file 3 "/usr/include/x86_64-linux-gnu/bits/types.h"
62      .file 4 "/usr/include/libio.h"
63      .file 5 "/usr/include/stdio.h"
64      .file 6 "/usr/include/malloc.h"

```

A listagem em assembly é exatamente a mesma obtida com a opção `-S`. No entanto, a listagem do código original também é apresentada e de uma forma bem confusa (repare nos números das linhas!), já que a enfase é no código em assembly... As otimizações feitas pelo compilador tendem a tornar a leitura do código em C meio esquisita. Pelo menos temos comentários dizendo o que está sendo feito com o código e os  $\mu_{op}s$ .

Neste ponto você pode se perguntar qual é a diferença de usar `-march=native`, afinal de contas? Em certos casos o compilador pode escolher usar, por exemplo, os registradores YMM (512 bits, ou 32 bytes). Se tivéssemos mais um valor de 64 bits na estrutura e quiséssemos zerá-la completamente, o compilador poderia escolher fazer algo assim:

```

vpxor xmm0, xmm0, xmm0
vmovdqu [rdi], ymm0
vzeroupper

```

Ao invés de algo mais tradicional:

```

xor eax,eax
mov [rdi],rax
mov [rdi+8],rax
mov [rdi+16],rax
mov [rdi+24],rax

```

## **Sobre os endereços em listagens em assembly obtidas com objdump ou GCC**

Considere o programinha abaixo:

```

/* test.c */
int x = 3;

int f(int a) { return a*x; }

```

Ao compilá-lo e obtermos a listagem em assembly (com os códigos em hexadecimal), obtemos algo assim:

```

$ gcc -O3 -masm=native -c test.c -o test.o
$ objdump -d -M intel test.o
...
Disassembly of section .text:
0000000000000000 <f>:

```

```

0: 8b 05 00 00 00 00      mov    eax, DWORD PTR [rip+0x0]
6: 0f af c7              imul   eax,edi
9: c3

```

Marquei em vermelho o pedaço da instrução MOV problemática... Essa instrução está, claramente, obtendo o valor da variável *x*, que foi declarada como global... Mas, como é que o compilador sabe que o endereço dela é 0x0?

Ele não sabe! Para esse módulo (este arquivo objeto) a variável global *x* é colocada no segmento *.data* logo no início, ou seja, no offset 0... Por isso a listagem coloca o offset 0 na instrução. Isso pode ser observado com:

```

$ objdump -t test.o
test.o:           file format elf64-x86-64
SYMBOL TABLE:
0000000000000000 l  df  *ABS*      0000000000000000 test.c
0000000000000000 l  d  .text       0000000000000000 .text
0000000000000000 l  d  .data       0000000000000000 .data
0000000000000000 l  d  .bss       0000000000000000 .bss
0000000000000000 l  d  .text.unlikely 0000000000000000 .text.unlikely
0000000000000000 l  d  .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 l  d  .eh_frame   0000000000000000 .eh_frame
0000000000000000 l  d  .comment    0000000000000000 .comment
0000000000000000 g  F  .text       0000000000000000 f
0000000000000000 g  O  .data       0000000000000004 x

```

Nessa listagem da tabela de símbolos temos, em cada linha, o endereço virtual do símbolo, um flag que diz se o símbolo é “local” (l) ou “global” (g), outro flag que diz se o símbolo é para o “debugger” (d), uma referência a um arquivo (f), uma função (F) ou outra coisa (O – mas pode ser um espaço também)... Segue a *section* onde o símbolo se encontra, o tamanho do símbolo em bytes e, finalmente, o nome.

Note que *x* é um símbolo “genérico” (do tipo “O”) que se encontra no offset 0, é global, localiza-se no segmento *.data* e tem 4 bytes de tamanho. Mas esse offset é relativo ao segmento *.data* deste módulo (test.o).

Por esse motivo, os endereços contidos em listagem assembly (geradas pelo compilador) devem ser entendidos como sendo *relativos* ao módulo ao qual pertence. É trabalho do **linker** atribuir um offset condizente com o programa final **depois** de agrupar todos os segmentos dos módulos.

## **Usando o make: Fazendo um bolo, usando receitas**

Ao invés de compilar seus códigos um a um, chamando o gcc e o nasm várias vezes, pode-se criar uma “receita” para cozinhar o bolo todo de uma maneira mais “automática”. Isso é feito com o utilitário **make**.

Com ele você cria um arquivo que contém, literalmente, receitas. Elas são compostas de regras e ações. Você só precisa encadear as regras para que uma dependa da outra e as ações serão feitas na ordem correta. Um exemplo, para compilar o famoso *helloworld.c*:

```

helloworld: helloworld.o
    gcc -o $@ $^

helloworld.o: helloworld.c
    gcc -c -o $@ $<

```

**Atenção:** A linha de comando tem que começar com um caractere ‘t’ (um “tab”).

Aqui temos duas receitas. A primeira receita é a mais importante porque estabelece a cadeia de dependências... No exemplo acima, a primeira receita nos diz que *helloworld* depende de *helloworld.o*. Se *helloworld.o* existir então, para fazer *helloworld*, a linha de comando contendo o gcc será chamada para linkar o arquivo objeto, criando o executável. Os macros \$@ e \$^ são

atalhos para o alvo (\$@) e as dependências (\$^). O make substituirá esses macros com os nomes de arquivos corretos, contidos na receita.

Note que a primeira receita depende de *helloworld.o*. A segunda receita nos diz como ele é feito. Ele depende de *helloworld.c*. A linha de comando abaixo dessa receita “faz” o *helloworld.o*, que é a dependência de *helloworld*.

A diferença entre \$< e \$^ é que o primeiro contém os nomes das dependências separadamente. O segundo contém todas as dependências numa única string. Isso é útil num *Makefile* assim:

```
test: test1.o test2.o test3.o
      gcc -o $@ $^

%.o: %.c
      gcc -c -o $@ $<
```

Aqui, todo arquivo com extensão '.o' será “feito” a partir de um arquivo correspondente com extensão '.c'. Ou seja, o gcc será chamado várias vezes, uma para cada arquivo '.c' no diretório correto. Uma vez que tenhamos 'teste1.o', 'teste2.o' e 'teste3.o', o gcc será chamado, de novo, para linkar os 3 objetos.

Outra coisa interessante do make é que as linhas de comando das receitas podem ser omitidas. O utilitário sabe o que fazer, na maioria dos casos. Se criássemos algo como:

```
test: test1.o test2.o test3.o
      gcc -o $@ $^

%.o: %.c
```

A sintaxe '%.o' e '%.c' é o equivalente do *wildcard* '\*.o' e '\*.c'.

No script acima, o compilador gcc será chamado com a opção -c para criar os objetos que dependem dos módulos em C. Para customizar esse comportamento podemos alterar algumas variáveis de ambiente. A variável CC nos diz qual é o compilador C que vai ser usado. Por default, make usa 'cc', que é um link simbólico para gcc.

As opções de compilação estão na variável CFLAGS. Assim, se quiséssemos compilar nossos códigos com a máxima otimização, poderíamos fazer:

```
CFLAGS += -O3 -march=native
OBJECTS = test1.o test2.o test3.o

test: $(OBJECTS)
      $(CC) -o $@ $^

%.o: %.c
```

A lista de objetos do qual 'test' depende foi colocada numa variável, bem como as opções '-O3 -march=native' foram adicionadas à variável preexistente CFLAGS. Ao usar \$(var) fazemos uma substituição literal da variável no comando ou na receita.

O make também permite receitas que tenham como alvo um falso arquivo. Por exemplo:

```
CFLAGS += -O3 -march=native
OBJECTS = test1.o test2.o test3.o

test: $(OBJECTS)
      $(CC) -o $@ $^

%.o: %.c

clean:
      rm *.o
```

O alvo *clean* não é a receita principal e não está na lista de dependências. Ele jamais será executado quando chamarmos *make* sem parâmetros. Mas, podemos chamar *make* passando o alvo desejado:

```
$ make clean
```

É prudente informar ao *make* que esse é um alvo “de araque” (*phony target*) usando a diretiva **.PHONY**:

```
CFLAGS += -O3 -march=native
OBJECTS = test1.o test2.o test3.o

# Diz ao make que 'clean' é um alvo 'impostor'.
.PHONY: clean

test: $(OBJECTS)
    $(CC) -o $@ $^

%.o: %.c

clean:
    rm *.o
```

Esses *phony targets* são úteis para realizarmos operações diferentes e até mesmo quanto nosso script exige múltiplos alvos. Lembre-se que a primeira receita é a que estabelece toda a cadeia de dependências. Assim, se tivermos que compilar um 'hello' e um 'bye', por exemplo, poderíamos fazer algo assim:

```
# Diz ao make que 'clean' é um alvo 'impostor'.
.PHONY: all clean

# Esta é a receita principal. Ela estabelece a dependência
# das receitas 'hello' e 'bye'.
all: hello bye

hello: hello.o
    $(CC) -o $@ $^

bye: bye.o
    $(CC) -o $@ $^

%.o: %.c

clean:
    rm *.o
```

A receita principal, 'all', estabelece que 'hello' e 'bye' têm que ser construídos primeiro. Cada um deles depende de um arquivo objeto que, por sua vez, são construídos pela receita '%.o: %.c'. A receita 'clean' está ai para permitir fazer uma limpeza do diretório onde os arquivos foram compilados.

Quanto as receitas em si, as ações podem ter mais que uma linha. Cada linha é executada num shell diferente. Por exemplo, se tivéssemos a receita para 'hello' assim:

```
hello: hello.o
    $(CC) -o $@ $^ -lm
    strip -s $@
```

O linker será chamado num shell e o 'strip' em outro, na sequência. Isso pode ser problemático se você quiser, por uma questão de estilo, usar mais que uma linha. No *make*, usar várias linhas para serem interpretadas pelo mesmo shell exige o uso do caractere '\n' no final da linha. Ainda, já que *make* tem macros começando com \$, se seu comando usa variáveis de ambiente você deve fazer as substituições usando \$\$:

```
hello: hello.o
    $(CC) -o $@ $^ -lm
    # As linhas abaixo são executadas num mesmo shell.
```

```
if [ -f $@ ]; then \
    echo Arquivo $@ compilado com sucesso em $$PWD. \
fi
```



# Capítulo 7: Medindo performance

Existem algumas interpretações para a palavra “performance”, aplicáveis ao contexto da execução de software. A única interpretação que interessa aqui é sinônima de “velocidade”.

Ao dizer “velocidade” você pode pensar isso como o número de instruções que o processador executará numa unidade de tempo ou, ainda, quanto tempo o processador leva para executar um conjunto de instruções. É uma aproximação lógica. Infelizmente, é impraticável.

## Ciclos de máquina e ciclos de clock

Todo o tempo gasto na execução de uma única instrução é composto de um conjunto de ciclos chamado de “ciclos de máquina”. Um ciclo de máquina é, por sua vez, composto de um conjunto de ciclos de clock e um “ciclo de clock” é período da forma de onda quadrada usada como base de temporização do processador. São os famosos *giga-hertz* da sua CPU. No meu caso, meu processador é um *i7-4770* com 3.4 GHz de velocidade, o que significa que cada ciclo de clock acontece em cerca de 0,29 ns<sup>40</sup>.

Um ciclo de máquina, em média, toma cerca de 4 ciclos de clock<sup>41</sup>. Embora, hoje em dia, a performance seja medida em termos de ciclos de clock porque os processadores modernos multiplicam o clock, internamente, continuarei usando o termo “ciclos de máquina” no decorrer deste texto.

Instruções diferentes gastam quantidades diferentes de ciclos de máquina. Instruções simples costumam gastar apenas um ciclo de máquina. Instruções mais complicadas (como CALL), gastam alguns ciclos (que pode chegar a algumas dezenas de ciclos).

## Contar ciclos de máquina não é fácil

Antigamente conseguíamos calcular, de antemão, quantos ciclos de máquina uma rotina gastaria. Bastava contar os ciclos de máquina de instruções individuais, já que elas eram lidas, decodificadas e executadas uma depois da outra, numa sequência bem definida.

A Intel usou um conceito de *arquitetura superescalar* que foi implementada nos抗igos processadores da série 29000 da AMD e nos seus próprios processadores da série i960 para acelerar o ciclo de leitura, decodificação e execução. O termo “escalar” aqui é sinônimo de “sequencial”... O “super” indica que os processadores, desde então, realizam os processos busca, decodificação e execução de instruções em paralelo. Enquanto uma instrução está sendo lida da memória (do cache, no caso) outra está sendo decodificada e outra está sendo executada...

A coisa complicou ainda mais, nas novas arquiteturas, com a reordenação automática de instruções. Esses novos processadores podem mudar a ordem do ciclo de execuções para garantir que a menor quantidade de ciclos de máquina seja gasto...

Temos outros fatores que afetam essa contagem de ciclos: Emparelhamento de instruções, reordenação de acessos à memória, efeitos relativos ao cache, threads, paginação etc. Tudo isso pode atrasar bastante a execução de instruções.

A ocorrência da maioria dessas interferências são difíceis de prever. Isso deixa a “contagem de ciclos” fora de questão. As únicas coisas que podemos fazer é: usar a intuição e medir o tempo!

40 Um nanosegundo equivale à bilionésima parte de um segundo.

41 Isso é um chute muito mal chutado... É uma base de comparação muito fraca, mas útil, para meus propósitos.

Intuição, neste caso, é olhar para uma rotina (em assembly) e avaliar, aproximadamente, o tempo gasto com base na complexidade das instruções. Isso é útil ao analisar ou desenvolver um rascunho da rotina desejada.

Como em qualquer coisa, nossa intuição pode estar totalmente errada. Precisamos de evidências. E a maneira mais simples de sabermos quantos ciclos, em média, uma rotina vai gastar é medindo. Nesse capítulo te mostro como medir esse gasto e como calcular o ganho ou perda de performance de uma rotina.

## Como medir?

Para determinar a velocidade de uma função é necessário medir a quantidade de ciclos de máquina que estão sendo gastos. Existem duas maneiras de fazermos isso:

1. Via hardware: Usando equipamentos como ICES (In Circuit Emulators) e/ou Analisadores Lógicos;
2. Via software: Usando contadores internos, RTC (Real Time Clock – o relógio do computador) ou algum *profiler* especializado como o *Intel VTune*.

ICES<sup>42</sup> e Analisadores Lógicos são caros e exigem conhecimento e experiência com eletrônica. Para a maioria de nós, pobres mortais, esses equipamentos têm curso proibitivos.

Nos processadores atuais existem recursos de medição de performance chamados de *performance counters*. Só que eles só estão disponíveis para o código executado pelo kernel. Nossas aplicações, no *userspace*, não têm acesso a esses contadores.

Felizmente, desde os processadores Pentium, existe um contador que está disponível em todos os privilégios de execução. Trata-se do *Timestamp Counter* (TSC). Ele conta quantos ciclos de clock já aconteceram desde o último *reset* (ou *power up*) do processador.

Para acessar o conteúdo desse contador basta executar a instrução RDTSC. Ela devolve a contagem no par de registradores EDX:EAX. Mesmo no modo 64 bits esse par de registradores é usado ao invés de apenas RAX.

Para ter uma ideia do valor que pode ser mantido pelo contador, se um processador funciona com clock de 3 GHz, podemos obter, sem perigo de *overflow*, quase 195 anos de contagem desde o momento que o processador foi colocado em funcionamento!

O exemplo abaixo mostra como obter o valor, usando a função intrínseca `_rdtsc`:

```
/* rdtsc1.c */
#include <stdio.h>
#include <x86intrin.h>

/* _rdtsc() tem o protótipo:
   unsigned long long _rdtsc(void); */

int main(int argc, char *argv[])
{
    printf("Ciclos: %lu\n", __rdtsc());
    return 0;
}
-----%<---- corte aqui -----%>-----
$ gcc -O3 -o rdtsc1 rdtsc1.c
$ ./rdtsc1
Ciclos: 82384266127
```

É garantido que a instrução RDTSC retorne um valor único, toda vez que for chamada (mesmo que

---

<sup>42</sup> Existe uma instrução **não documentada** nos processadores Intel (mas, documentada nos processadores AMD) chamada IceBP. Trata-se de um “breakpoint” por hardware, mas nem vale a pena estudá-la!

seja chamada em threads diferentes):

## Aumentando a precisão da medida

Os processadores modernos tendem a executar instruções “fora de ordem”. Pode parecer estranho, já que o conceito de um programa é, justamente, execução de instruções de forma sequencial, uma depois da outra. Só que, algumas vezes, não faz muita diferença, do ponto de vista funcional, que uma sequência de instruções seja alterada. E essa modificação de ordem pode resultar num aumento considerável de performance. Eis um exemplo. Considere o pequeno fragmento de código abaixo:

```
loop:  
    mov al,[rsi]  
    mov [rdi],al ; dependência de AL, acima.  
    inc rdi  
    inc rsi  
    dec rcx  
    jnz loop  
...
```

Lembre-se que podemos ter duas ou mais instruções sendo executadas “ao mesmo tempo” no mesmo processador lógico. No exemplo acima é óbvio que as duas primeiras instruções terão que ser executadas sequencialmente porque a segunda depende da atualização de AL, feita na primeira instrução. O processador, esperto como é, poderá reordenar as instruções assim:

```
loop:  
    mov al,[rsi]  
    inc rsi      ; Repare que essa instrução foi 'reordenada'.  
    mov [rdi],al  
    inc rdi  
    dec rcx  
    jnz loop
```

Se o processador tem a capacidade de executar duas instruções simultaneamente<sup>43</sup>, é evidente que o segundo loop gasta 3 ciclos de máquina, enquanto o primeiro executa em 4. Isso nos dá um aumento de performance de 25% (ou seja, o segundo loop é executado em 75% do tempo do primeiro).

Já que instruções podem ser reordenadas na tentativa de aumentar a performance, o código sob teste pode ser reordenado, inclusive colocando instruções **antes** ou **depois** das leituras do contador de ciclos de clock. É evidente que isso nos dará medida errada do gasto de ciclos daquilo que queremos testar. Felizmente, podemos usar instruções que *serializam* o processador. O termo *serializar* significa que o processador esperará que todas as instruções pendentes sejam executadas e, só então, continuará o processamento.

A maioria das instruções que serializam o processador só podem ser executadas em níveis altos de privilégio (ring 0), mas existem algumas que são úteis no *userspace*. É o caso de instruções como CPUID, LFENCE, SFENCE e MFENCE. A instrução CPUID também serializa o processador.

As instruções LFENCE, SFENCE e MFENCE são interessantes: Elas são chamadas de instruções de barreira (uma “cerca”, *fence* em inglês, é uma “barreira” que colocamos em torno de uma casa) porque esperam que o processador termine de executar **todas** as instruções pendentes que fazem carga (Load), armazenamento (Storage) ou ambas as coisas (M, de MFENCE significa *Memory*).

Para garantir que os efeitos de execução fora de ordem não interfiram em nossas medidas, precisamos serializar o processador. A rotina de leitura do TSC, em forma de macro, ficaria assim:

---

43 Processadores baseados em arquiteturas mais recentes – Haswell, por exemplo – têm o potencial de executarem até 8 instruções simultaneamente!

```

/* A variável 'x', passada para esse macro, deve ser
   do tipo 'unsigned long long'. */
#define TSC_READ(x) \
{ \
    register unsigned int lo, hi; \
    \
    __asm__ __volatile__ ( \
        "mfence;" \
        "rdtsc;" \
        : "=a" (lo), "=d" (hi) ); \
    \
    (x) = ((unsigned long)hi << 32) | \
          (unsigned long)lo; \
}

```

No macro acima não há perigo de misturar bits entre os valores das variáveis *lo* e *hi*, já que os bits superiores estarão zerados (de novo: RDTSC só atualiza EAX e EDX, mas zera a porção superior de RAX e RDX). Só que esse cálculo adicional pode muito bem ser colocado dentro do bloco assembly e, se usarmos uma variável *long* como retorno, é garantido que RAX vai ser colocado nela:

```

#define TSC_READ(x) \
{ \
    register unsigned long r; \
    \
    __asm__ __volatile__ ( \
        "mfence;" \
        "rdtsc;" \
        "shll $32,%edx;" \
        "orl %edx,%eax" \
        : "=a" (r) : : "%rdx" ); \
}

```

Como RDTSC altera, inclusive RDX, é necessário colocá-lo na lista dos registradores preservados para dar uma chance ao GCC de salvá-lo, se necessário.

Essas instruções extras, SHL e OR, serão reordenadas depois do MFENCE, é claro, mas elas provavelmente serão executadas em paralelo com a instrução CALL da chamada a ser testada. Mesmo que a chamada seja colocada *inline* em seu código, essas instruções adicionais serão emparelhadas com outras, tendo muito pouca influência no valor final. E, mesmo que tenha, elas gastarão apenas 1 ciclo de clock. Não se trata de grandes perdas...

MFENCE é uma instrução do SSE2. Se seu processador for um antigo Pentium III ou se não suportar SSE2, você poderá querer trocá-la por CPUID, mas é necessário fazer algumas modificações, já que CPUID altera EAX, EBX, ECX e EDX, precisamos colocar RBX e RCX na lista de preservação (RDX também, mas por causa de RDTSC):

```

#define TSC_READ(x) \
{ \
    register unsigned long r; \
    \
    __asm__ __volatile__ ( \
        "xor %eax, %eax;" \
        "cpuid;" \
        "rdtsc;" \
        "shll $32,%edx;" \
        "orl %edx,%eax" \
        : "=a" (r) : : "%rbx", "%rcx", "%rdx" ); \
}

```

Existe um *paper* da Intel mostrando que esse simples uso de leitura do TSC pode ser problemático quando se mede a performance de uma função... O problema é que a segunda chamada gastará tempo executando MFENCE e os MOVs finais poderão estar fora de ordem. A Intel recomenda formas de leitura diferentes. Uma para o início da medição e outra para o fim:

```
unsigned long r0, r1;
```

```

__asm__ __volatile__ (
    "mfence;"
    "rdtsc;"
    "shll $32,%%rdx;"
    "orl %%rdx,%%rax"
    : "=a" (r0) : : "%rdx"
);

/* Função a ser medida... */
f();

__asm__ __volatile__ (
    "rdtscp;"
    "shll $32,%%rdx;"
    "orl %%rdx,%%rax"
    : "=a" (r1) : : "%rdx", "%rcx"
);

/* tsc conterá a diferença dos timestamps. */
tsc = r1 - r0;

```

O segundo bloco em assembly usa a instrução RDTSCP que serializa o processador sem o overhead da chamada de MFENCE.

### **Mas, o gcc possui funções “intrínsecas” para executar CPUID e RDTSC!**

Sim, possui... Elas estão localizadas nos headers *ia32intrin.h* e *cpuid.h*, mas como são chamadas individuais é possível que o compilador gere mais código do que é necessário ou menor do que queremos. Um exemplo de uso, onde obteremos valores mais ou menos iguais aos que conseguiríamos usando a função em assembly, é este:

```

#include <x86intrin.h>

...
unsigned long c, c0;
int dummy;

_mm_mfence();
c0 = __rdtsc();
f();
/* __rdtscp() usa um ponteiro para obter o valor do registrador IA32_TSC_AUX_MSR. */
c = __rdtscp(&dummy);
c -= c0;
...
-----%<---- corte aqui -----%>-----
; código parcial gerado. Compilado com -O2
...
mfence
rdtsc
mov rbx,rax
sal rdx,32
or  rbx,rdx

call f

rdtscp
sal rdx,32
or  rax,rdx
sub rax,rbx

; Neste ponto, RAX é o conteúdo da variável c.

```

Que é um código bem decente...

Só tome cuidado com as otimizações do compilador. Se for usar o máximo de otimizações, você pode topar com o rearranjo de código que não medirá coisa alguma (as leituras do TSC podem ser rearranjadas). Para evitar isso, recomendo que compile o código a ser testado em um módulo

separado e use a opção de otimização '-O3'. Já o código que contém as chamadas às funções intrínsecas `_mm_fence`, `__rdtsc` e `__rdtscp`, num módulo com a otimização '-O0'.

## Melhorando a medição de performance

Quando você brincar um bocado com a medição de ciclos de clock perceberá que, para uma mesma função sob teste, a contagem de ciclos é diferente em cada medida. Isso é perfeitamente explicável graças ao trabalho feito pelo processador ao gerenciar tarefas, páginas, cache, interrupções etc. Ou seja, o valor que você está medindo nunca será exato<sup>44</sup>.

Como precisamos obter um significado a partir dos valores lidos e não temos exatidão, nada mais justo que usar o recurso da *estatística*. E o meio mais simples de obter significado de um conjunto de valores é olhar para a média. Se tenho 50 valores ligeiramente diferentes, posso dizer que um valor único representando a média desses valores é aquilo que procuro.

Para funções pequenas, uma maneira de obter uma boa média é medir a execução de diversas chamadas à mesma função. A medição anterior poderia ser feita assim:

```
...
unsigned long c0, c1;
int dummy;

_mm_mfence();
c0 = __rdtsc();
/* Função f() executada 50 vezes! */
f(); f(); f(); f(); f(); f(); f(); f(); f();
f(); f(); f(); f(); f(); f(); f(); f(); f();
f(); f(); f(); f(); f(); f(); f(); f(); f();
f(); f(); f(); f(); f(); f(); f(); f(); f();
f(); f(); f(); f(); f(); f(); f(); f(); f();
c2 = __rdtscp(&dummy);

/* Imprime a média simples de 50 medidas! */
printf("Ciclos de clock: %lu\n", ((c2 - c1) / 50));
...
```

Substituir as 50 chamadas explícitas acima por um loop não faz o que parece:

```
for (i = 50; i > 0; --i)
    f();
```

O loop não será “desenrolado”. Pelo menos não totalmente! É mais provável que você acabe com um código assim:

```
    mov  ebx, 50
.L1:
    call f
    dec  ebx
    jg   .L1
...
```

Ao invés de 50 chamadas você acabará adicionando uma inicialização (de EBX), 50 decrementos e 49 saltos condicionais, no melhor dos casos... Assim, sua medição não será **somente** da função, mas do código de controle do loop também. E, assim, você dá adeus à precisão que queria na medição!

## O cálculo do ganho de performance

Para permanecermos na mesma sintonia, o cálculo do ganho de performance sempre é feito em relação a alguma rotina original que, supostamente, é mais lenta que a rotina otimizada. A relação é esta:

---

44 No [apêndice B](#) mostro um jeito de aumentar a precisão das medidas.

$$G[\%] = \left( \frac{F_{original} - F_{otimizada}}{F_{otimizada}} \right) \cdot 100$$

Onde  $G$  é o ganho percentual da rotina otimizada em relação à rotina original.

Suponha que a função original gaste 1000 ciclos de clock e que a rotina otimizada gaste 100. Temos:

$$\begin{aligned} F_{original} &= 1000 \\ F_{otimizada} &= 100 \end{aligned}$$

$$G = \left( \frac{1000 - 100}{100} \right) \cdot 100 = \left( \frac{900}{100} \right) \cdot 100 = 900\%$$

Ou seja, temos um ganho de performance na função otimizada de 900% em relação à função original (ou, vendo de outra forma,  $F_{original}$  é 10 vezes mais lenta!).

E se tivéssemos o contrário? Se a rotina “otimizada” gastasse mais ciclos que a rotina “original”? Suponha que  $F_{original}$  gastasse 100 ciclos e  $F_{otimizada}$ , 125. O valor de  $G$  será de -25%, que é exatamente a **perda** de performance da rotina “otimizada” (note o valor negativo!):

$$\begin{aligned} F_{original} &= 100 \\ F_{otimizada} &= 125 \end{aligned}$$

$$G = \left( \frac{100 - 125}{100} \right) \cdot 100 = \left( \frac{-25}{100} \right) \cdot 100 = -25\%$$

É claro que, para uma medida estatística mais precisa, teríamos que realizar diversas medições, obter o média dos valores e calcular o erro relativo. Por exemplo. Se obtenho 1300 e 1280 ciclos de duas medições então ela provavelmenet gasta  $1290 \pm 0.78\%$  ciclos:

$$Valor_{medio} = \frac{1300 - 1280}{2} = 1290$$

$$erro = \pm \frac{1300 - 1290}{1290} \cdot 100 \approx 0,78\%$$

Acontece que essa variação (0,78%) é tão pequena que não vale o esforço. Erros menores que uns 2% podem ser desconsiderados... Mas, suponha que tenhamos 3 medidas: 1300, 200 e 800. O valor médio será de cerca de 767 e o erro relativo será de  $\pm 69,5\%$ . Um erro demasiadamente grande.

Neste caso vale a pena usar a média<sup>45</sup>...

Toda vez que formos comparar performance essa é a regra do jogo: Medimos a quantidade de ciclos de clock de duas rotinas, a original e a otimizada, e aplicamos a formula acima para obter o ganho ou a perda. Fazmos isso com vários medidas e usamos a média para nos orientarmos, considerando a faixa de erro das medições...

## **Quando um ganho “vale à pena”?**

Você verá, algumas vezes, que descarto valores pequenos de ganho como “desprezíveis”. Por exemplo, se uma rotina original gasta 120 ciclos e uma otimizada gasta 115, o ganho de performance é de, aproximadamente, 4.3%. O valor percentual é pequeno, mas não é somente ele o avaliado. O detalhe é que houve um ganho de apenas 5 ciclos de clock!

Se o ganho relativo for pequeno e o ganho absoluto também for, então descarto esse suposto ganho, considerando-o desprezível. Isso é diferente, por exemplo, se tivéssemos  $F_{original}=11000$  e  $F_{otimizada}=10300$ . Isso nos dá um ganho relativo de 6.8% – que é pequeno – mas, em valores absolutos, ganhamos 700 ciclos de clock. Neste último caso a performance relativa não impressiona

---

<sup>45</sup> Pode ser interessante empregar uma análise estatística mais refinada, usando *desvio padrão* para determinar o quanto os valores desviam-se da média aritmética simples.

mesmo, mas o ganho absoluto pode ser interessante, no caso dessa rotina otimizada estar sendo usada dentro de um loop.

Os dois valores têm que ser pesados e, também, as circunstâncias... Não vale à pena usar uma versão “otimizada” de uma rotina que será executada apenas uma vez quando o ganho relativo é pequeno, mas vale à pena usá-la quando o ganho absoluto é “grande” e essa rotina é usada dentro de um loop ou é chamada diversas vezes.

## Usando “perf” para medir performance

Linux disponibiliza uma ferramenta, por linha de comando, que permite usar diversos *performance counters*. Além dos ciclos gastos por uma rotina, você pode medir *cache misses*, *page misses*, erros na predição dos *branches* etc. O nosso código de teste ficaria simplesmente assim:

```
/* test.c */
extern void f(void); /* rotina sob teste. */

int main(int argc, char *argv[])
{
    f();
    return 0;
}
```

Ao usar *perf* você não mede a performance de uma rotina específica, mas do seu programa como um todo. Ao compilar e linkar (sem usar o *tsc.asm*), obtemos o programa 'test'. Daí podemos usar:

```
$ perf stat -r 20 ./test
Performance counter stats for './test' (20 runs):

      0,143854 task-clock (msec)          #    0,599 CPUs utilized          ( +-  0,72% )
          0 context-switches             #    0,348 K/sec                  ( +-100,00% )
          0 cpu-migrations              #    0,000 K/sec
         117 page-faults               #    0,813 M/sec
     489.242 cycles                  #    3,401 GHz                   ( +-  0,80% )
<not supported> stalled-cycles-frontend
<not supported> stalled-cycles-backend
        407.411 instructions          #    0,83  insns per cycle       ( +-  0,30% )
        77.121 branches                #  536,103 M/sec                 ( +-  0,23% )
        2.401 branch-misses            #    3,11% of all branches       ( +-  4,01% )

  0,000240241 seconds time elapsed          ( +-  1,05% )
```

O programa 'test', executado 20 vezes (opção '-r 20'), gastou em média 482242 ciclos de clock, executou 407411 instruções, fez 77121 saltos. Nesse percurso, que durou apenas 240 µs, ocorreram 117 *page faults* e 2401 *branch misses*. Mas, note, foi o programa como um todo, incluindo as rotinas de inicialização e finalização da *libc*.

É claro que você pode tentar descontar os valores obtidos por um programa que não faz nada. Mas, um programa que não faz nada não tem muitas inicializações e finalizações para fazer, tem?

O utilitário *perf* é muito bom para dar uma idéia do que está acontecendo. Estamos interessados na performance de rotinas, não do programa geral. Isso significa que o método usando TSC\_READ é mais útil, neste contexto. Mesmo sendo mais impreciso...

# Capítulo 8: Otimizações “automáticas”

Compiladores C e C++ modernos conseguem melhorar muito o código criado pelo programador usando uma série de algoritmos de otimização, reorganizando o código final. Aqui eu vou te mostrar algumas das otimizações “automáticas” realizadas por esses compiladores.

## Níveis de otimização

Existem dúzias de otimizações que podem ser habilitadas no compilador. Entender e consultar toda as possibilidades é uma tarefa cansativa. Para evitar o cansaço o compilador disponibiliza 6 conjuntos de otimizações, habilitadas pelas opções '-O0', '-O1', '-Os', '-O2', '-O3', '-Ofast'. As opções estão listadas aqui na ordem em que geram código do menos otimizado ao mais otimizado.

A opção '-O0' criará o código mais “puro” possível. Ao contrário do que parece, esse nível **não** significa que nenhuma otimização será feita. '-O0' realiza otimizações **mínimas** no seu código. Isso tende a gerar funções maiores e menos performáticas e, ao mesmo tempo, faz com que o compilador traduza o seu código-fonte quase que ao pé da letra. Ele é útil quando for debugar código. Também é útil quando você construir códigos de teste para medir a velocidade, via TSC\_READ. Essa opção evita que o compilador faça com que parte do seu código desapareça, devido às otimizações.

As opções '-O1', '-O2' e '-O3' habilitam otimizações adicionais, progressivamente mais agressivas. Todas elas habilitam, por exemplo, otimizações de *dead code elimination* (dce), *data store elimination* (dse), *guess branch probability* (onde o compilador tenta “adivinar” se um salto condicional será feito ou não – útil para otimizar ‘if’s e loops). Mas a opção '-O2' habilita otimizações de alocação de registradores, alinhamento de loops, alinhamento de saltos e *global common subexpression elimination*. A opção '-O3' habilita as otimizações mais agressivas, incluindo vetorização e funções inline.

A opção '-Os' é a mesma coisa que a opção '-O2', mas ela diz ao compilador que temos preferência por gerar códigos pequenos ('s' de *small*). Essa opção **não** gera os menores códigos possíveis, ela só é uma dica para que o compilador escolha instruções ou conjuntos de instruções mais simples e menores que as escolhidas pela otimização '-O2'.

A opção '-Ofast' é a opção '-O3' com algumas adições ainda mais agressivas.

A opção '-O', sem o valor numérico, é sinônima de '-O1'.

## “Common Subexpression Elimination” (CSE)

Essa é uma das otimizações mais básicas e mais interessantes. O nome dessa técnica é bem evidente. Trata-se de eliminar expressões comuns. Suponha que seu programa tenha algo assim:

```
x = 2 * a + 3 * b + c;  
y = 2 * a + 3 * b + d;
```

Repare que parte das duas expressões têm, em comum, a sub-expressão “ $2 * a + 3 * b$ ”. Ao habilitar CSE o compilador criará código equivalente a isto:

```
_tmp = 2 * a + 3 * b;  
x = _tmp + c;  
y = _tmp + d;
```

O ganho de performance é óbvio: Ao invés de quatro multiplicações e quatro adições, teremos duas multiplicações e três adições. O ganho de performance pode chegar a perto de 100%! Outro efeito colateral é a potencial diminuição do código.

CSE agressivo pode ser um problema ao usar nossa rotina de obtenção do TSC. Em alguns casos o compilador pode entender que a chamada à TSC\_READ e o cálculo envolvendo a mesma variável é uma CSE e simplesmente eliminar uma das chamadas. Por isso uso a opção -O1 ou -O2, ou até '-O0', na compilação do código testador.

## Desenrolamento de loops

Outra otimização simples. Se você precisa chamar uma função quatro vezes, para facilitar a codificação você poderia fazer algo assim:

```
for (i = 0; i < 4; i++)
    DoSomething();
```

Ao ver algo assim o compilador poderá desenrolar esse loop, eliminando-o completamente, e gerar quatro chamadas para *DoSomething()*. Este é o cenário mais óbvio. No entanto, o compilador pode ser mais esperto com loops com mais iterações. Suponha que ao invés de 4 tenhamos 400. O compilador poderá escolher fazer um desenrolamento parcial. Algo mais ou menos assim:

```
/* Suponha que esse seja o loop original. */
for (i = 0; i < 400; i++)
    DoSomething();

/* O compilador poderia substituir por isso: */
for (i = 0; i < 50; i++)
{
    DoSomething();
    DoSomething();
    DoSomething();
    DoSomething();
    DoSomething();
    DoSomething();
    DoSomething();
    DoSomething();
}
```

Aqui o compilador escolheu continuar com um loop, para poupar o cache L1, mas desenrolá-lo para que tenhamos menos saltos condicionais. Desenrolamento de loops pode tornar seu código maior e não melhorar tanto assim a performance, por isso o compilador tem um limite para o desenrolamento. O limite é definido de acordo com a arquitetura e parâmetros internos do compilador (mas pode ser ajustado)...

Todas as opções de otimização, exceto pelo nível 0, habilitam alguma forma de *loop unrolling*.

## Movendo código invariante de dentro de loops (*Loop Invariant Code Motion*)

Alguns códigos podem ser rearranjados drasticamente para evitar processamentos desnecessários. Considere o seguinte:

```
for (i = 0; i < 100; i++)
{
    x = a + b;
    DoSomething(x);
}
```

A variável 'x' é calculada 100 vezes considerando os mesmos valores contidos nas variáveis 'a' e 'b'. Mas 'a' e 'b' não são atualizadas dentro do loop... Poderíamos mover a linha onde 'x' é calculado para fora do loop sem nenhum problema. E é isso que o compilador fará:

```

x = a + b;
for (i = 0; i < 100; i++)
    DoSomething(x);

```

## **Eliminação de código morto (Dead Code Elimination)**

Às vezes criamos rotinas que simplesmente jamais serão executadas. Ou, quando o são, não fazem coisa alguma. O compilador tentará eliminar esses códigos. Eis alguns exemplos:

```

for (i = 0; i < 10000; i++)      /* loop provavelmente será eliminado. /
if (k && !k)                  /* DoSomething() jamais será chamado. */
    DoSomething();
while (1)
    DoSomething();
    DoSomethingElse();          /* DoSomethingElse() jamais será chamado. */

```

Outro exemplo que você pode encontrar é o uso de funções que não retornam valor algum, não alteram variáveis globais. O compilador pode eliminar essas funções em alguns casos. Por exemplo:

```

#include <stdio.h>
int f(int x, int y) { return x*y; }
void main(void) { f(10, 20); puts("ok"); }

```

Quando usamos uma opção de otimização diferente de -O0 o compilador vai ser livrar da chamada para *f* sem pestanejar.

Ainda outro exemplo de código morto é o assinalamento de uma variável para si mesma. É um macete útil para evitar avisos de “parâmetros não usados” em suas rotinas. Por exemplo, o código abaixo talvez te dê esse tipo de aviso:

```
int f(int x, int y) { return x+x; }
```

Se você realmente quer que a função acima recebe dois parâmetros, mas não use um deles, evite o aviso fazendo assim:

```
int f(int x, int y) { y=y; return x+x; }
```

A atribuição de *y* para si mesmo será eliminada pelo compilador e ele não pode reclamar que você não usou *y*...

## **Eliminação de armazenamento morto (Dead Store Elimination)**

Do mesmo jeito que código morto é descartado na otimização, o não uso de variáveis declaradas também faz com que elas sejam descartadas, quando possível. Mas, o compilador pode te avisar disso...

## **Previsão de saltos (Branch Prediction)**

Desde o 486 os processadores Intel têm um recurso chamado *branch prediction*. Graças à natureza superescalar do processador (execução de diversas instruções simultaneamente, na mesma pipeline), quando o ele encontra um salto condicional, é possível que o estado atual dos flags não reflita o estado em que estarão quando a instrução for executada. Quer dizer: Antes dos 486 toda instrução de salto serializava o processador, colocando uma unidade de execução em estado *idle* (de espera) até que todas as instruções antes do salto fossem executadas. Só então poderia haver certeza do

estado dos flags.

Na arquitetura superescalar o processador usa um recurso estatístico para tentar “prever” se um salto condicional será feito ou não.

O compilador tenta dar uma mãozinha tentando prever se um salto vai ser sempre feito ou não. Quando escrevemos algo assim:

```
if (condicao)
{
    ...
}
```

O compilador tende a gerar um código mais ou menos assim:

```
cmp [condicao], 0
je .L1
...
.L1:
```

Note que o salto é feito se a comparação for **falsa**! Mas, para aproveitar o *branch prediction* do processador, em loops, o compilador tende a modificar a posição onde a condição de parada de um loop é feita. Por exemplo:

```
...
while (condicao)
{
    ...
}
-----%<---- corte aqui ----%<-----
; código que você espera que o compilador crie:
    cmp [condicao], 0
    je .L1
.L2:
    ...
    jmp .L2
.L1:
-----%<---- corte aqui ----%<-----
; código gerado pelo compilador:
    jmp .L1
.L2:
    ...
.L1:
    cmp [condicao], 0
    jne .L2
```

Neste caso é esperado que a comparação seja verdadeira para cada iteração do loop. Assim, se o compilador não tentasse adivinhar se saltos serão ou não tomados, misturar loops e 'if's causaria uma confusão dos diabos nos algoritmos de *branch prediction* do processador, tornando o código mais lento.

Essa “adivinhação” não é perfeita. Podemos dar uma mãozinha ao compilador usando uma função “embutida” chamada `__builtin_expect`. Ela é usada para melhorar a adivinhação do compilador consideravelmente. Como exemplo, suponha que esperemos que uma condição de um *if* seja **falsa** na maioria das vezes. Poderíamos escrever algo assim:

```
if (__builtin_expect(x <= 0, 0))
{
    ...
}
```

Se soubermos que 'x' será positivo ou zero na maioria das vezes, o código acima ajudará o compilador a organizar os saltos condicionais para aproveitar, ao máximo, o *branch prediction*.

O kernel usa duas macros que, por sua vez, usam a função `__builtin_expect`. Trata-se de *likely* e *unlikely*:

```
#define likely(c) __builtin_expect(!!(c), 1)
#define unlikely(c) __builtin_expect(!!(c), 0)
```

Esses nomes são mais intuitivos... *likely* pode ser traduzido para “possivelmente”...

## Simplificações lógicas

O que você espera que o compilador faça com expressões como esta?

```
y = a | (a & b);
```

Não é evidente, mas a expressão acima é simplificada para, simplesmente, “y = a;”. O GCC (e também outros bons compiladores) realizam facilmente as seguintes simplificações lógicas:

Expressão completa	Expressão simplificada
a & 0	0
a & ~0	a
a   0	a
a   ~0	~0
a & a	a
a   a	a
a   (a & b)	a
(a & b)   (a & ~b)	a
(a & b)   (a & c)	a & (b   c)
(a & b)   (~a & c)   (b & c)	a ? b : c
(a & ~b)   (~a & b)	a ^ b

Tabela 8: Simplificações lógicas feitas pelo compilador.

Essas e outras simplificações são esperadas do ponto de vista do compilador. Elas não são válidas apenas para expressões de atribuição, mas também acontecem com os operadores lógicos booleanos (|| e &&). Por exemplo, se tivermos:

```
if (((a < 0) && (b < 0) && (c < 0)) ||
    ((a < 0) && (b >= 0) && (c < 0)) ||
    ((a >= 0) && (b < 0) && (c < 0)) ||
    ((a >= 0) && (b >= 0) && (c < 0)))
DoSomething();
```

Esse monte de comparações será simplificado, pelo compilador, para:

```
if (c < 0)
DoSomething();
```

Este exemplo parece óbvio e esse recurso é bem interessante. Mas ele pode ser problemático se você estiver lidando com dispositivos externos. Por exemplo: Uma das rotinas do meu antigo curso de assembly tinha, numa das listagens, algo assim:

```
*bitplane |= 0;
```

O objetivo era ler o conteúdo da memória de vídeo, fazer um OR com todos os bits zerados e gravar o conteúdo na memória de vídeo novamente. Isso pode ser feito numa única instrução:

```
or byte [bitplane],0
```

Isso era um passo necessário para atualizar os *latches* dos *bitplanes* da memória de vídeo. Acontece que com as otimizações ligadas, o compilador vai ignorar a linha acima... Afinal, fazer um OR com zero é a mesma coisa que não fazer OR algum!

Casos como esse – e ainda bem que são raríssimos – são sérios candidatos ao desenvolvimento em assembly, não em C!

## Simplificação de funções recursivas

O GCC tende a eliminar as chamadas recursivas automaticamente. O código abaixo é famoso. Trata-se do cálculo de fatorial:

```
unsigned long factorial(unsigned long x)
{
    if (x <= 0) return 1;
    return x*fatorial(x-1);
}
```

Sem otimizações o compilador criará código como o mostrado abaixo:

```
fatorial:
    sub  rsp,8

    mov  [rsp],rdi          ; usa a pilha para armazenamento temporário.
    cmp  qword [rsp],0
    jg   .L2
    mov  rax,1
    jmp  .L3

.L2:
    mov  rax,[rsp]
    sub  rax,1
    mov  rdi,rax
    call factorial          ; Eis a chamada recursiva.
    imul rax,[rsp]

.L3:
    add  rsp,8
    ret
```

O problema com a recursividade é que ela coloca pressão sobre a pilha. Cada chamada à *fatorial*, acima, usa 16 bytes da pilha: Oito bytes para o parâmetro, e oito para o endereço de retorno, colocado lá pela instrução CALL. Se passarmos um parâmetro com valor gigantesco, logo teremos uma exceção de *Stack Overflow* nas mãos, por causa dos muitos empilhamentos.

O código gerado com a máxima otimização não tem quaisquer chamadas recursivas. O compilador transforma o código original em algo assim:

```
long factorial(long x)
{
    long r = 1;

    while (x > 1)
        r *= x--;

    return r;
}
```

É claro que certas recursividades não podem facilmente ser simplificadas. De fato, algumas têm simplificações matematicamente impossíveis. Nesses casos o compilador não tem o que fazer a não ser incluir a chamada recursiva. Vale a pena dar uma olhada no código gerado, em assembly gerado pelo GCC...

## Auto vetorização (SSE)

No modo x86-64, SIMD (SSE) está sempre disponível. Este é o motivo dos registradores de XMM0 até XMM7 serem usados como parâmetros de entrada de valores em ponto flutuante, na convenção de chamada. Mas, o compilador tende a usar apenas o primeiro componente desses registradores. SSE suporta agrupar até 4 *floats* num mesmo registrador.

Realizar 4 operações com floats (ou 'ints') ao mesmo tempo é o que chamamos de “vetorização”.

Em certos casos o compilador decide usar vetorização para tornar o código mais eficiente. Eis um exemplo:

```
extern int a[256], b[256], c[256];

void f(void)
{
    int i;

    for (i = 0; i < 256; i++)
        a[i] = b[i] + c[i];
}
```

Já que temos 256 inteiros em cada um dos arrays, o código gerado tenderá a usar vetorização com valores inteiros (32 bits):

```
f:
xor eax, eax
align 4
.L2:
movdqa xmm0,[b+rax]
paddd xmm0,[c+rax]
movdqa [a+rax],xmm0
add rax,16
cmp rax,1024
jne .L2
rep ret
```

Isso equivale, mais ou menos, ao código, em C<sup>46</sup>:

```
extern int a[256], b[256], c[256];

void f(void)
{
    __m128i *pa = (__m128i *)a,
            *pb = (__m128i *)b,
            *pc = (__m128i *)c;

    for (i = 0; i < 64; i++)
        *pa++ = _mm_add_epi32(*pb++, *pc++);
}
```

Mas, atenção! O compilador não tentará vetorizar referências a arrays se o tamanho do loop não for conhecido de antemão. A função abaixo não tende a ser vetorizada:

```
void f2(int *out, int *a, int *b, size_t count)
{
    while (count--)
        out[count] = a[count] + b[count];
}
```

O compilador não tem como saber qual é o valor de 'count' de antemão!

## Otimizações através de profiling

Todas as otimizações do compilador são feitas de maneira estática, sem levar em conta a execução

---

46 Ver capítulo “Instruções Extendidas”.

*de facto* do código gerado. O compilador tenta usar esse conjunto de regras para gerar o código mais performático possível, de acordo com o nível de otimização selecionado, mas nem sempre consegue!

O GCC possui o recurso de “gravar” informações sobre a execução do código gerado para que possamos compilar o código uma segunda vez levando em conta essas informações. É uma compilação de dois passos: Primeiro, compilamos o código com as otimizações selecionadas e geramos um executável. Depois, executamos o código e um arquivo com extensão “.gcda” é criado, contendo informações sobre saltos e outros detalhes. Ao compilar uma segunda vez, usando este arquivo, o compilador terá uma base melhor para decidir reorganizar o código.

Usar o recurso de *branch profiling* é simples assim:

```
$ gcc -O3 -march=native -fprofile-generate -o test test.c
$ ./test
...
$ ls -l
-rwxr-xr-x 1 user user 20200 Jan  6 11:37 test
-rw-r--r-- 1 user user   113 Jan  6 11:37 test.c
-rw-r--r-- 1 user user   140 Jan  6 11:42 test.gcda
$ gcc -O3 -march=native -fprofile-use -o test test.c
$ ls -l
-rwxr-xr-x 1 user user 6810  Jan  6 11:44 test
-rw-r--r-- 1 user user   113 Jan  6 11:37 test.c
-rw-r--r-- 1 user user   140 Jan  6 11:42 test.gcda
```

A opção '-fprofile-generate' injeta código para colher informações do seu programa e armazená-lo no arquivo '.gcda' e, por isso, o executável ficará bem mais que deveria. Ao compilar pela segunda vez, usando a opção '-fprofile-use' o compilador usa essas informações e otimiza melhor o código gerado. Ambas as opções aceitam parâmetros informando o nome do arquivo '.gcda' que, por default, tem o mesmo nome do arquivo '.c'.

# Capítulo 9: Caches

Este capítulo contém informações conceituais sobre o funcionamento do cache que não condizem totalmente à realidade. Deixei de fora o conceito de *set associativity* porque creio que ele é supérfluo para o desenvolvedor de software. Trata-se de detalhe interno do processador que, ao meu ver, interessa apenas ao fabricante. Deixando esse conceito de fora, o que resta serve perfeitamente para os propósitos do entendimento das vantagens e limitações impostas pelos caches.

*Cache* é uma estrutura usada para conter, dentro do processador, um pedaço de um subconjunto de dados originalmente contidos na memória RAM. O exagero do “pedaço do subconjunto” é proposital. Quero dizer que apenas um pedaço pequeno da memória é copiado para os caches.

A existência dos caches deve-se ao fato de que a memória RAM é lenta. E tem que ser, já que memórias rápidas são muito caras. Os caches, então, são artifícios usados para aumentar a velocidade de acesso aos dados que o processador manipula. Para fazer isso, esse acesso nunca é feito diretamente na memória RAM. Tem que, obrigatoriamente, passar pelos caches.

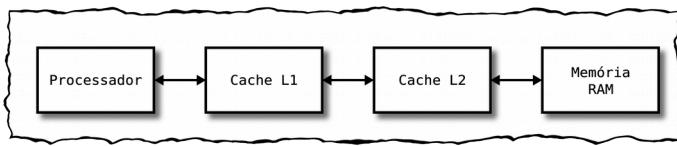


Figura 10: Acesso à memória sempre é feita pelos caches.

Processadores modernos implementam vários níveis de *cache*. O *cache L1* (*Level 1*) é aquele mais próximo aos circuitos de manipulação de dados do processador. Ele é dividido em dois tipos: L1d e L1i, com 32 KiB cada. Os sufixos 'd' e 'i' especificam, respectivamente, *data* e *instruction*. Isso significa que o *cache L1*, na verdade, são dois: Um dedicado para dados e outro para instruções.

Existe um segundo nível (L2) de maior capacidade, mas ele mistura tudo o que está na memória: código e dados. Não há divisão. O motivo da existência do segundo nível é que o primeiro é pequeno e, por isso, precisa ser recarregado frequentemente. O *cache L2* é o cache do cache, ele foi criado para evitar a recarga do cache L1 diretamente a partir da memória RAM, mais lenta. Níveis maiores podem ser encontrados em processadores mais modernos. Não é incomum encontrar um cache L3 e arquiteturas como *Haswell* suportam até um cache L4...

Do ponto de vista da performance, precisamos melhorar ao máximo o acesso feito ao cache L1, tanto o L1d, quanto o L1i. Não há muita necessidade de nos preocuparmos com o L2 porque ele é grande: tem uns 256 KiB. Por isso o restante do capítulo tratará o cache como se tivesse apenas um nível.

## A estrutura dos caches

Diferente da memória do sistema, que é organizada de maneira linear com um byte atrás do outro, a memória do cache é organizada em “linhas”. Cada linha tem a estrutura que contém bits de controle, usados internamente pelo processador. Dentre outras coisas eles determinam a validade da linha... Contém também uma “etiqueta” ou “tag”, seguida de 32 a 64 bytes de dados, dependendo da arquitetura do processador<sup>47</sup>.

47 Vou lidar com linhas de tamanho de 64 bytes daqui pra frente. Processadores como i5 e i7 têm caches com linhas desse tamanho, normalmente.

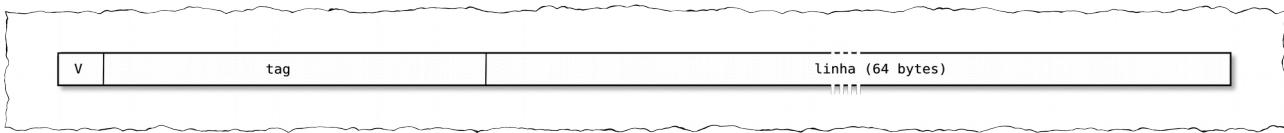


Figura 11: Estrutura de uma linha do cache L1.

Essa “tag” nada mais é do que os bits superiores **do endereço físico**<sup>48</sup> associado a uma linha. O processador usa essa tag para determinar se um bloco de memória está contido no cache ou não.



Figura 12: Estrutura de um endereço linear no contexto do cache L1.

A figura anterior mostra como um endereço físico é interpretado pelo cache. É claro que o offset tem apenas 6 bits de tamanho ( $2^6=64$ ). O campo “linha” especifica uma das 512 linhas do cache. E o campo “tag” é a identificação que será comparada com a respectiva linha no cache.

Se o endereço nos dá o número da linha no cache, ao tentar acessar memória, a primeira coisa que o processador fará é determinar se a linha é válida (ela pode ainda não ter sido “usada”!) e contém a mesma tag do endereço. Caso a tag exista no cache, se a linha for inválida, ela será carregada do cache L2 (a mesma coisa acontece do cache L2 para o L3, se esse existir, e do L3 para a memória do sistema).

Se a linha for “válida”, mas a tag não for a mesma, então a linha precisará ser “trocada”. O conteúdo atual será escrito no cache de nível superior e depois os 64 bytes correspondentes à nova tag serão lidos. De novo, o processo é repetido nos caches de nível superior...

Cada leitura e escrita de linhas do cache L1 gasta de 1 a 4 ciclos de clock, dependendo do processador, já para os caches de nível superior a quantidade de ciclos gastos aumenta (uns 12 ciclos na arquitetura *Haswell*). E se esse processo de validação de linhas ocorre sempre que o processador tentar ler/escrever num endereço de memória, fica evidente que se tivermos algum meio de manter linhas de cache válidas e carregadas, menos tempo o processador terá que gastar fazendo trocas de linhas.

Cada processador lógico nos núcleos tem seu próprio par de caches, L1i e L1d. Já os caches L2 e L3 (se existir um) são compartilhados por todos.

## Determinando o tamanho de uma linha

Pode ser útil o seu programa saber o tamanho de uma linha do cache L1. Para obtermos essa informação basta usar a instrução CPUID:

```
#include <stdio.h>
#include <cpuid.h>

int main(int argc, char argv[])
{
    unsigned int a, b, c, d;

    /* A operação 0x01 com CPUID nos dá informações sobre
       o cache L1. */
    __cpuid(1, a, b, c, d);
```

48 Veja sobre “endereço linear” no capítulo sobre “memória virtual”.

```

/* bits 15~8 de EBX, multiplicados por 8
   dão o tamanho da linha de cache L1. */
printf("L1 cache line size = %u bytes).\n",
      (b & 0xff00) >> 5);

return 0;
}

```

## O dilema do alinhamento

Tentar conter dados e código em uma única linha de cache é, obviamente, um fator decisivo para garantir performance. Se seus dados ultrapassam o limite de uma linha, corre o risco das cópias entre caches e a memória física acontecerem. A mesma coisa vale para as instruções em linguagem de máquina.

Uma maneira de ter certeza de que uma linha está devidamente preenchida é contarmos a quantidade de bytes do código ou dos dados a partir do início de uma linha. Só que isso é impraticável! Concorda comigo que ficar contando os bytes para cada instrução em “linguagem de máquina” é tarefa que deixaria qualquer um meio maluco? Além do mais, se tivermos que lidar com nossos programas ao nível de “linguagem de máquina” (micro-códigos), então qual é a utilidade de uma linguagem de programação? E, mesmo que você tenha essa paciência toda, o desperdício de memória pode ser grande na tentativa de evitar o *overlap* entre linhas.

**Overlap?** Imagine que a última instrução numa linha de cache tenha 5 bytes de tamanho, mas os primeiros 60 bytes já estejam em uso na linha. Ao colocar essa instrução de 5 bytes, 4 deles ficariam na linha válida do cache e o byte excedente terá que ficar numa outra linha que pode ou não estar carregada no cache...

Você poderia preencher os 4 bytes finais dessa linha com NOPs e colocar a próxima instrução na linha seguinte, mas isso causará o mesmo problema e criará outro: Teremos instruções inúteis que gastam tempo!

Com dados ocorre o mesmo, mas é mais controlável. Suponha que seu código use uma estrutura assim:

```

struct mystruct_s {
    long a[7];           /* 56 bytes: do offset 0 até 55. */
    char b[5];           /* 5 bytes: do offset 56 até 60. */
    long d;              /* 8 bytes: do offset 61 até 68. (ops! Cruza duas linhas!) */
};

```

Ao declarar uma variável desse tipo, se ela estiver alinhada com o início de uma linha, o membro de dados 'd' ainda assim cruzaria a fronteira entre **duas** linhas. Neste caso a solução é simples. Basta alterar a ordem das variáveis na estrutura:

```

struct mystruct_s {
    long a[7];           /* 56 bytes: do offset 0 até 55. (linha 1) */
    long d;              /* 8 bytes: do offset 56 até 63. (linha 1) */
    char b[5];           /* 5 bytes: do offset 58 até 62. (linha 2) */
};

```

Vamos usar duas linhas de qualquer jeito, mas ao não colocarmos uma variável parcialmente em cada uma, quando uma delas for “trocada”, a outra poderá continuar válida.

Então, o dilema é este: Sempre que for possível, precisamos colocar instruções inteiras e dados inteiros dentro de linhas de cache, evitando que eles cruzem a fronteira entre duas delas. Um jeito de conseguir isso é usando alinhamento. Se alinharmos o início de nossas funções de 64 em 64 bytes garantiríamos que, pelo menos, o início da função está no início de uma linha do cache L1i.

Essa forma de alinhamento também é impraticável porque causará mais trocas de linhas dos caches

ao longo prazo. Só temos 512 linhas disponíveis no L1 e, hoje em dia, código e dados podem ser muito grandes...

Uma solução de bom senso poderia ser o alinhamento por QWORD. Ele não desperdiça tanto o espaço da linha e é útil, na arquitetura x86-64, onde ler um QWORD desalinhado, da memória para um registrador de uso geral, causa uma penalidade de 1 ciclo de clock... Mas isso não é tão útil no que se refere às instruções. Uma única instrução em linguagem de máquina pode ocupar até 15 bytes (esse é o limite imposto pela Intel e pela AMD). Assim, no caso dos códigos, é mais útil o alinhamento de 16 bytes.

Isso não significa que você não possa usar alinhamentos menores, como uma espécie de ajuste fino... O compilador C tenta alinhar loops e inícios de funções tanto quanto possível, uma vez que ele sabe onde os pontos críticos estarão. No caso de assembly, podemos alinhar nossas funções e até mesmo loops usando a diretiva **align** do NASM:

```
align 16      ; entrada da função alinhada!
f:
    xor rax, rax
    mov rcx, 1000
    jmp .L1

    ; Já que o loop ficará retornando a esse ponto, alinhamos por QWORD!
    ; Se o salto condicional estiver muito longe (digamos, uns 128 bytes além
    ; deste ponto) é melhor alinharmos por 16 bytes também!
    align 8
.L2:
    ...
.L1:
    dec   rcx
    jnz   .L2
```

No exemplo acima usamos dois alinhamentos diferentes. O início da função estará alinhada com o início de uma linha do cache L1i, mas a posição de retorno do loop pode não precisar disso. Podemos alinhá-lo por QWORD se o loop inteiro couber dentro de algumas poucas linhas do cache.

Com os dados o compilador tende a alinhar o início de uma variável por QWORD, mas nem sempre consegue (como é o caso das estruturas), se bem que isso é característica “dependente de implementação”, de acordo com a especificação da linguagem C. Existem exceções: Ao usar SIMD os tipos `_m64`, `_m128`, `_m256` e `_m512` são, automaticamente, alinhados por QWORD (`_m64`), a cada 16 bytes (`_m128`), 32 (`_m256`) e 64 bytes (`_m512`). Esse último tipo, é claro, cabe numa linha inteira! Esses tipos são forçosamente alinhados para garantir a boa performance de instruções de carga a partir da memória, caso contrário o processador gastará mais ciclos de clock, independente do tipo ter cruzado a fronteira de uma linha ou não. Se, além do desalinhamento, o processador tiver que cruzar a fronteira entre linhas de cache, então muito mais ciclos serão adicionados à execução da instrução e, em alguns casos, uma exceção do tipo *segmentatil fault* pode ocorrer.

### **Dica para usar melhor os caches...**

Mantenha seus códigos e dados pequenos! Se puder encurtar seus loops para que caibam em uma linha de cache, tanto melhor. Senão, procure não consumir muitas linhas. Lembre-se que, além do seu programa, existe o kernel, device drivers, scripts etc, sendo executados!

Um código que tem um loop com tamanho de mais de 4 KiB<sup>49</sup> pode causar sérios problemas de performance. Neste caso, 64 linhas de cache terão que estar válidas para todo o loop. Mas, dentro do loop podemos ter chamadas para outras funções, o que pode invalidar parte dessas linhas, causando a perda de centenas ou milhares de ciclos de clock.

---

49 O motivo dos 4 KiB ficarão claros no capítulo sobre “memória virtual”.

Esteja ciente para o fato de que essa dica é impraticável na maioria das vezes, mas é importante que seja seguida sempre que possível. Na prática você “queimarará a mufa” em rotinas que eleger como críticas e resolver desenvolvê-las em assembly.

### **Audaciosamente indo onde um byte jamais esteve...**

Por causa das frequentes trocas de linhas entre os caches e a memória, outro conceito importante quando lidamos com os caches é o de **temporalidade**. Você pode encarar o funcionamento dos caches sob o aspecto **espacial** (o tamanho de uma linha) e **temporal** (o tempo em que os dados ficarão sob os cuidados do cache). Quando a AMD e a Intel incorporaram SIMD nos seus processadores, começaram também a preocuparem-se em controlar a temporalidade de blocos de dados contidos nos caches.

Ao aumentar o tamanho dos registradores que precisam acessar variáveis alinhadas, a “pressão” nos caches aumenta consideravelmente. Pode ser útil dizer ao processador que uma linha deve ser mantida válida por mais tempo que o necessário, evitando a recarga do cache cada vez que um bloco de 16 bytes (SSE) seja lido ou gravado. Outra maneira de encarar a temporalidade é o quanto “fácil” o processador pode “esquecer” uma linha de cache.

O “ajuste fino” da temporalidade é feito em todos os níveis de cache ao mesmo tempo. As instruções *PREFETCHxx* são usadas para dar uma “dica” ao processador (que pode respeitá-la ou não) sobre a temporalidade de uma linha e, de lambuja, carregará (*prefetch*) a linha no cache L1. Existem 3 instruções desse tipo:

Instrução	Significado
PREFETCHTx	Onde $x$ é um valor entre 0 e 3.  Quanto menor o valor de $x$ , menos “esquecível” é a linha. O mnemônico pode ser entendido como <i>prefetch t#</i> . Onde Tx é o nível de temporalidade.
PREFETCHNTA	A parte “NTA” significa “Non-Temporal Access”. Esse PREFETCH põe a <b>dica</b> de que a linha não pode ser facilmente “esquecida”.
PREFETCHW	O “W” diz que a linha será cacheada com a <b>dica</b> de que o dado está sempre pronto para ser “escrito”. Esse modo invalida os caches de alta ordem para a linha, fazendo com que ela precise ser escrita nesses níveis quando a linha for “esquecida”.

É necessário tomar cuidado com a instrução PREFETCH... Embora o ajuste fino seja útil em certas circunstâncias, evitando gastos de ciclos de clock na troca de linhas, isso pode colocar grande “pressão” no cache L1.

### **Funções *inline* e a saturação do cache**

Quando alguém aprende sobre funções *inline* nas linguagens C e C++ tende a usá-las para ganhar performance. Afinal, chamadas e retorno de funções gastam uns 20 ciclos de clock. O problema de abusar das funções inline é justamente a saturação do cache L1i.

Sempre que uma função inline é usada, o código da chamada, que era um simples CALL, é substituído pela incorporação do código dessa função inline na função chamadora. Isso tem o potencial de tornar seu código muito grande e códigos grandes violam o princípio “mantenha seus códigos pequenos”.

Vale a pena perder algumas dezenas de ciclos com o par de instruções CALL/RET do que usar funções inline que podem consumir milhares de ciclos adicionais por causa do cache saturado!

Repare que, em C, existem duas maneiras de tornar uma função “inline”:

1. Através do uso do atributo “inline” na assinatura da função – que é apenas uma dica ao compilador que pode ignorá-la;
2. Colocando chamadas a funções declaradas no mesmo módulo.

Para evitar o segundo caso basta declarar o protótipo da função no módulo chamador e declarar a definição da função em outro:

```
/* func.c */  
  
/* A função f é DEFINIDA neste módulo. */  
int f(int x) { return x + x; }  
-----%<----- corte aqui -----%<-----  
/* main.c */  
  
/* A função f é somente DECLARADA neste módulo.  
   Nunca será “inline”, mesmo se usarmos otimização -O3! */  
extern int f(int);  
  
int g(int x)  
{ return f(x) + 2; }
```

# Capítulo 10: Memória Virtual

Há anos vi uma boa analogia a respeito de “memória virtual”: Um palhaço tenta equilibrar três ou mais bolinhas coloridas, enquanto as joga no ar. Duas delas estarão o tempo todo nas mãos dele, mas as outras sempre estarão suspensas. Eventualmente uma das bolas que estava nas mãos do palhaço irá voar e uma que estava voando cairá em uma das mãos. O palhaço não é capaz de trabalhar com todas as bolinhas ao mesmo tempo, então uma delas terá que ser “jogada” e recuperada depois.

Fica parecendo que o palhaço está “equilibrando” três bolinhas nesse ato de malabarismo quando, na realidade, ele está lidando sempre com duas. É mais ou menos assim que funciona o mecanismo de *page swapping*. Enquanto o processador lida com uma porção da memória, outra pode não estar fisicamente presente.

## Virtualização de memória

Memória é endereçada como se fosse um grande array. Um endereço é um índice para esse array. Quando dizemos que a memória é virtual, dizemos que um endereço não aponta mais para a memória física, mas aponta para um modelo conceitual. Ou seja, um endereço pode apontar para memória que não existe.

Neste modelo conceitual a memória parece linear, composta de blocos adjacentes (um bloco depois do outro), do mesmo jeito que a analogia do array, mas o endereço que usamos para acessar dados nesse modelo conceitual, chamado de *endereço linear*, precisa ser **traduzido**, através de um **mapeamento**, para um endereço usado pela memória física (que não é “conceitual”, mas existe de fato!). Esse modelo conceitual é chamado de *virtual address space*.

## Espaços de endereçamento

Um espaço de endereçamento é um modelo conceitual de como um endereço é interpretado. Basicamente, existem 3 deles, nos modos i386 e x86-64:

- **Physical Address Space:** O endereço é o mesmo que aparece no barramento de endereços do processador. É o que é usado para ler o conteúdo de um pente de memória.
- **Logical Address Space:** O endereço é obtido através do endereço base, vindo de um descritor de segmento através de um registrador seletor de segmento, que é adicionado a um offset. A não ser que estejamos lidando com memória virtual, este espaço é, essencialmente, o mesmo que o espaço físico.
- **Linear Address Space:** Neste espaço um endereço é interpretado como um valor inteiro que, decomposto, contém índices para tabelas de tradução de página e um offset. Esse endereço é conhecido como *endereço linear*.

Já que estamos lidando com o espaço de endereçamento virtual aqui, daqui para frente falarei apenas de endereços lineares e físicos.

## Paginação

Você não lê um livro de uma vez só, lê? Fazer isso é um desafio até pra aquelas pessoas com domínio absoluto em “leitura dinâmica”. Por isso um livro é dividido em capítulos, sessões, tópicos, páginas, parágrafos, sentenças, palavras e letras (e números, e símbolos, ...). Essa divisão estrutural

faz com que você possa passar de um item a outro sem que tenha que visualizá-los na totalidade.

A analogia é boa com aquilo que seu processador faz com o acesso à memória... O *virtual address space* é a totalidade teórica da memória que pode ser usada (o livro, na analogia) e essa memória é dividida em páginas.

Considere uma página como sendo um bloco com tamanho de 4 KiB. Quero dizer, o processador, se fizer uso do recurso da paginação, ele dividirá o *virtual address space* nesses bloquinhos. Cada página é mapeada num conjunto de tabelas, e a manipulação dessas tabelas só é possível no *ring 0*, portanto, apenas o kernel tem condições de manipulá-las.

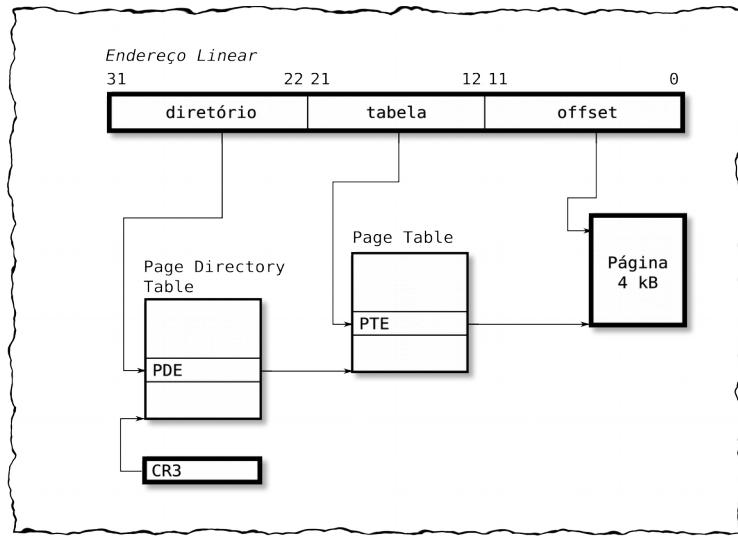


Figura 13: Endereço linear (32 bits)

Quando estamos lidando com o “modo paginado” do processador, o registrador de controle CR3 contém o endereço físico, na memória, de uma tabela de 4 KiB chamada *Page Directory Table* (PDT). Cada entrada dessa tabela contém o endereço físico de outras tabelas de 4 KiB chamadas *Page Table* (PT). Essas últimas contêm entradas com os endereços físicos de uma página, na memória física.

Repare na estruturação de um endereço linear: Os primeiros dois campos (“diretório” e “tabela”) são, na verdade, índices para entradas nas tabelas PDT e PT. O índice para a PDT nos fornece uma *entrada para o diretório de página* (*Page Directory Entry*, ou PDE). Da mesma forma, o índice para a PT nos fornece um PTE (*Page Table Entry*).

De posse do endereço base da página, obtida da PTE, adicionando o offset contido nos 12 bits inferiores do endereço linear, o processador obtém o endereço físico desejado.

Este esquema maluco, que lembra um pouco a estrutura de subdiretórios em *file systems*, nos dá algumas vantagens... As entradas das tabelas contém, além do endereço físico, alguns bits de controle: Além do privilégio da página (*supervisor* – correspondente ao *ring 0* – ou *user* – correspondente ao *ring 3*), temos também um bit indicando se a página está presente na memória física ou não. Se a página não estiver presente na memória física, de acordo com esse bit, então uma exceção de *page fault* é gerada, dando a chance ao processador de fazer algo a respeito (*page swapping*).

## Paginação e swapping

Paginação é uma coisa que existe desde a época dos famigerados *mainframes*. A ideia é, justamente, a de que memória física é um recurso caro e com pouca capacidade<sup>50</sup>. Nos antigos mainframes,

<sup>50</sup> No caso dos antigos *mainframes*, a capacidade de memória era de apenas alguns quilobytes (KiB!).

algumas dessas páginas eram armazenadas em “fita” e outras eram mapeadas na “memória física”. Sempre que quiséssemos usar uma página que não estava presente na memória, uma página “física” era gravada na “fita” e líamos outra página da fita que substituí a página previamente mapeada na “memória física”. Mais ou menos como a analogia das bolinhas coloridas do palhaço.

Até hoje é mais ou menos assim que a memória gerenciada pelo seu sistema operacional funciona. Só que ao invés de fita, as páginas podem ser armazenadas em disco: num arquivo ou numa partição. Essa “troca” do espaço físico da memória com o conteúdo do disco é conhecido como *page swapping* (troca de páginas).

Não vou descrever como a partição de swap (ou o arquivo de paginação, no caso do Windows) é estruturado aqui. Esse detalhe diz respeito somente ao sistema operacional e não tem qualquer relevância no que concerne à performance de nossas aplicações. O que importa é que *page swappings*, quando ocorrem, causam um tremendo atraso no processamento. O processador tem que, literalmente, parar tudo o que está fazendo, acessar o disco, modificar alguma coisa na estrutura das tabelas de páginas e só então retornar ao processamento normal.

### Tabelas de paginação no modo x86-64

No modo x86-64 essa estrutura de “subdiretórios” foi estendida: Cada tabela continua tendo, no máximo, 4 KiB, mas as entradas agora tem 64 bits de tamanho. Por causa disso cada tabela passa a ter 512 entradas e os índices no endereço linear têm 9 bits de tamanho. E, graças à necessidade de um maior espaço de endereçamento, ao invés de apenas duas tabelas passamos a ter quatro. A primeira é chamada PML4T (*Page Map Level 4 Table*). A segunda, PDPT (*Page Directory Pointer Table*), a terceira e a quarta são nossas velhas conhecidas PDT e PT.

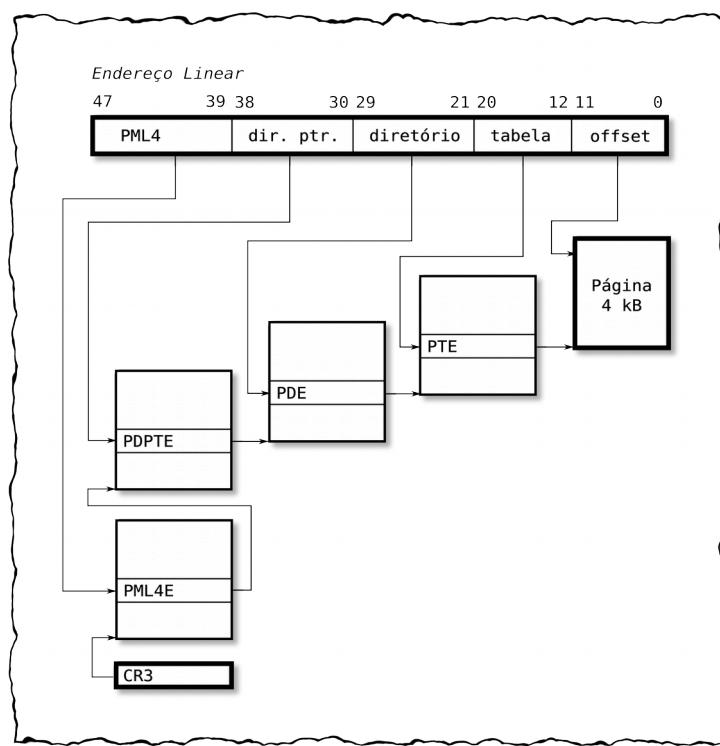


Figura 14: Endereço linear típico (64 bits).

Para facilitar a compreensão sobre essas tabelas, daqui por diante falarei apenas sobre a PT, que mapeia uma página diretamente. Toda a discussão aplica-se às outras tabelas, só que elas apontam para páginas que contém tabelas e não a página endereçável pelo componente *offset* do endereço linear (isso ficou claro, até aqui, creio).

## As entradas da Page Table

Cada entrada da PT (PTE) tem a seguinte estrutura:

```
struct table_entry_s __attribute__((packed))
{
    unsigned long p:1;           /* A página está presente? */
    unsigned long rw:1;          /* read-write ou read-only? */
    unsigned long us:1;          /* Privilégio: User ou Supervisor? */
    unsigned long pwt:1;         /* page write-through */
    unsigned long pcd:1;         /* page cache disable */
    unsigned long a:1;           /* acessada? (pode ser usada por software) */
    unsigned long d:1;           /* suja? (dirty?) (pode ser usada por software) */
    unsigned long ps:1;          /* PSE: Sempre 0 na PTE. */
    unsigned long g:1;           /* página global? */
    unsigned long unused1:3;     /* precisa ser zero! */
    unsigned long addr:40;        /* bits superiores do endereço físico. */
    unsigned long unused2:11;     /* precisa ser zero! */
    unsigned long xd:1;          /* NX bit: Se setado não permite código executável. */
}
```

Apenas alguns desses bits nos interessam: O bit P, se zerado, indica que qualquer tentativa de acesso a uma página para essa entrada causará um *page fault*. Neste caso, os demais bits da entrada não são considerados para coisa alguma. Os bits RW e US indicam, respectivamente, se a página pode ser lida e escrita (*read/write*) ou apenas lida (*read only*) e o privilégio necessário para acessá-la (*user* ou *supervisor*). As páginas relacionadas ao kernel são marcadas como *supervisor*.

O bit G indica que essa página precisa ser mantida no cache global de páginas (chamado de TLB) a todo custo, evitando que a tradução precise ser refeita a cada acesso à página. Isso poupa tempo. Falarei mais sobre TLBs adiante.

O bit XD indica se a página pode conter código executável ou não. No *userspace*, quando você aloca memória com *malloc*, o kernel “cria” entradas em tabelas de páginas para o seu processo com o bit XD setado (XD é sigla de *eXecution Disable*). Por isso não basta alocar memória, colocar um monte de bytes contendo um código em *linguagem de máquina* lá e saltar para alguma posição desse bloco alocado. Essa técnica de *code injection* não funciona e mostrarei como fazê-la corretamente mais adiante.

## As extensões PAE e PSE

Deu pra perceber que o esquema de paginação não “aumenta” a quantidade de memória diretamente acessível pela CPU? O endereço linear de 48 bits continua sendo traduzido para um endereço físico de 48 bits. Tudo o que a paginação faz é “quebrar” a memória em blocos de 4 KiB.

A partir dos processadores *Pentium Pro*, lançados em 1995, a Intel resolveu criar uma extensão que aumenta o tamanho de um endereço físico contido nas tabelas de páginas. No modo i386 o endereço físico, usado como base para uma página, passa a ter 36 bits de tamanho (tornando possível usar 64 GiB do *virtual address space*) e, no modo x86-64, passa a ter 52 bits (4 PB do *virtual address space*). Essa extensão, a *Physical Address Extension*, ou PAE, é ativada através do bit 5 do registrador de controle CR4.

O endereço linear ainda tem o mesmo tamanho. O que PAE permite é *mapear* uma página num espaço de endereçamento virtual maior, mas as páginas continuam com o mesmo tamanho (4 KiB) e, portanto, a faixa completa de todos os endereços lineares possíveis (de 0 até 0x0000ffffffffff, no modo x86-64) continua sendo de, no máximo, 256 TB (ou 4 GiB do modo i386).

Um detalhe interessante é que a extensão PAE têm que estar **obrigatoriamente** habilitada no modo x86-64.

Outra extensão é a PSE (*Page Size Extension*), introduzida pela Intel no *Pentium III*. O nome já diz:

Com essa extensão podemos usar páginas maiores que 4 KiB. O kernel do Linux usa PSE, se disponível. Com essa extensão podemos ter páginas de tamanho variável entre 4 KiB, 2 MiB e 1 GiB de tamanho, no modo x86-64.

PSE funciona assim: Um dos bits da estrutura de uma tabela de diretórios (exceto a PT!) é usado para habilitar esse tamanho maior que 4 KiB. O bit é nomeado de PS (*Page Size*). Ele serve para suprimir a tabela seguinte (PT, por exemplo), fazendo com que o offset no endereço linear aumente de tamanho.

No caso do x86-64, se o bit PS estiver ativo numa PDE (veja figura anterior) então a PDT torna-se a nova PT e o offset passa a ter 21 bits de tamanho (ou seja, uma página passa a ter 2 MiB de tamanho). Por outro lado, se PS estiver setado numa PDPT, então a PDPT torna-se a nova PT e o offset terá 30 bits de tamanho (ou seja, teremos uma página de 1 GiB de tamanho).

Além dos bits PS dessas tabelas de diretório, o bit 4 do registrador de controle CR4 também precisa estar setado para que a extensão tenha efeito.

### ***Translation Lookaside Buffers***

Se cada vez que o processador usar um endereço linear ele precisar traduzi-lo, então o processamento ficaria muito lento. Como existem os caches para poupar o processador o acesso à memória, diretamente, também existem caches para tradução de endereços lineares chamados *Translation Lookaside Buffers* (TLBs). Cada vez que acesso à memória é feito, uma comparação é feita com o conteúdo dos TLBs. Se a tradução já estiver disponível, nenhuma tradução nova é necessária (porque já foi feita!).

O processador tem diversos TLBs mantidos internamente. A quantidade e a especialização (TLBs para dados, código ou “compartilhados”) depende da arquitetura. Em meu processador, um *Core i7*, da arquitetura *Haswell*, existem 64 TLBs para dados (dTLBs) e 128 TLBs para código (iTTLBs). Onde cada processador lógico têm 16 iTLBs. No entanto, saber disso não é lá muito útil, já que não é possível ter acesso às TLBs diretamente. O processador as mantém para seu uso particular.

### ***A importância de conhecer o esquema de paginação***

*Page Faults*, *swapping*, TLBs... tudo isso tem impactos na performance...

Sabendo que o processador divide a memória física em unidades “atômicas” de tamanho mínimo de 4 KiB (páginas), podemos evitar *page misses* (e, portanto, *page swappings*) alocando memória em blocos de tamanho múltiplo de 4 KiB e alinhados com o início de uma página (os 12 bits inferiores do endereço linear zerados, ou seja *offset=0*). Isso evita, por exemplo, que nossos dados cruzem a fronteira entre páginas.

Se o offset do endereço linear base for zero e o bloco tem 4 KiB de tamanho, então estaremos necessariamente dentro de uma página. Se nossos dados puderem ser condidos de acordo com essa restrição, não há motivos para o processador gerar exceções do tipo *page fault* e realizar *page swappings*, assim, ganhamos tempo precioso em nossas rotinas!

A mesma lógica pode ser feita para um conjunto de páginas. Determinar quantas páginas ainda estão disponíveis na memória física, mas ainda não mapeadas, e com base nessa informação alocarmos somente o tamanho necessário evitará termos páginas flutuando “no ar” como se fossem as bolinhas coloridas na mão do proverbial palhaço...

Algumas dessas coisas são levadas em conta nas rotinas da *libc*.

## Tabelas de páginas usadas pelo userspace

Ao que parece, cada processo no *userspace* tem suas próprias tabelas de páginas, copiadas a partir de um processo pai, sendo *init* o patriarca. Todo processo novo surge a partir do *fork* de um processo pai, onde as tabelas de páginas do processo original são copiadas para o filho. Por isso ambos os processos compartilham todos os dados, incluindo descritores de arquivos...

No entanto, ao realizar o *fork*, o kernel ajusta o status das páginas de dados de ambos os processos como *read-only* e, quanto há uma tentativa de escrever numa variável, uma *page fault* ocorre, modificando o mapeamento da página do processo para uma outra página, com a respectiva cópia da página original... Esse procedimento é conhecido como *copy-on-write* (copia quando escreve), ou COW.

Ao que parece, ambos os processos compartilham do mesmo endereço linear, como pode ser demonstrado no código abaixo:

```
/* test.c */
#include <unistd.h>
#include <stdio.h>

int x = 0;

void main(void)
{
    pid_t pid;

    printf("Antes do fork - Processo pai: &x = 0x%016lx\n", &x);

    pid = fork();
    if (pid == -1)
    {
        puts("ERRO no fork().");
        return;
    }
    else if (pid == 0)
    {
        /* Processo filho */
        printf("Depois do fork - Processo filho: &x = 0x%016lx\n", &x);
        x = 1;
        printf("Depois do fork (escrita: x = %d) - Processo filho: &x = 0x%016lx\n", x, &x);
    }
    else
    {
        /* Processo pai */
        printf("Depois do fork - Processo pai: &x = 0x%016lx\n", &x);
        x = 2;
        printf("Depois do fork (escrita, x = %d) - Processo pai: &x = 0x%016lx\n", x, &x);
    }
}
-----%<---- corte aqui -----%>-----
$ gcc -o test test.c
$ ./test
Antes do fork - Processo pai: &x = 0x00000000000600b4c
Depois do fork - Processo pai: &x = 0x00000000000600b4c
Depois do fork (escrita, x = 2) - Processo pai: &x = 0x00000000000600b4c
Depois do fork - Processo filho: &x = 0x00000000000600b4c
Depois do fork (escrita: x = 1) - Processo filho: &x = 0x00000000000600b4c
```

Como você pode observar, tanto no processo pai quanto no processo filho o endereço linear da variável 'x' é 0x600b4c. Para processos diferentes isso só é possível se ambos os processos usarem tabelas de diretórios de páginas diferentes, já que em ambos os casos PDPTE=0, PDPE=3 e PTE=0, no endereço linear.

Isso significa que, embora os processos compartilhem o mesmo **endereço linear**, ao usar tabelas diferentes, eles **não** compartilham o mesmo **endereço físico**. Cada processo tem seu próprio conjunto de tabelas de páginas e, portanto, sempre que é feito um chaveamento de tarefas envolvendo processos diferentes, CR3 será carregado de acordo, pelo kernel.

De fato, no modo i386, o conteúdo de CR3 para uma tarefa é mantido na estrutura do TSS (*Task State Segment*), nos dizendo que CR3 é modificado, de fato, entre chaveamentos de contextos.

No modo x86-64 isso não é feito com assistência direta do processador, já que o TSS é bem diferente nesse modo.

Atualizar CR3 constantemente tem o danoso efeito colateral de invalidar as TLBs, exceto aquelas cujas entradas em tabelas de página estejam marcadas como “globais”.

## Alocando memória: *malloc* e *mmap*

A implementação da função *malloc* (e suas derivadas: *calloc* e *realloc*), na *libc*, é interessante: Ao alocar menos que 128 KiB de memória (32 páginas), *malloc* toma conta do espaço reservado no heap da aplicação. Provavelmente o loader do sistema operacional ou a própria *libc* pré alocam páginas suficientes para não ter que requerer remapeamento.

Para blocos menores que 128 KiB, *malloc* usa a system call *sbrk*, que modifica o tamanho da memória já alocada para a imagem binária alocada ao carregar a aplicação. Essa função aumenta a quantidade de páginas alocadas, tomando como base as que já estão lá. Já para blocos maiores que 128 KiB, *malloc* usa a system call *mmap*, que aloca páginas **privadas** (ao *userspace*).

Um algoritmo otimista é assumido para *malloc*, isto é, a *libc* supõe que a memória requisitada está disponível para alocação. Não há garantias que as páginas recém alocadas estejam presentes na memória física (possibilitando o *swapping*) e, se o processo requisitar memória que o kernel informa que não pode disponibilizar, *malloc* retornará um ponteiro nulo (NULL), deixando que seu código decida o que fazer.

Por causa do *virtual address space*, o retorno do ponteiro nulo é um evento raro, na maioria das vezes. Nem por isso você deverá ser leviano com o ponteiro retornado por *malloc*. É essencial **sempre** verificar-lo:

```
void *ptr;
if ((ptr = malloc(size)) == NULL)
{
    /* Oops! Um erro de alocação aqui! */
    ...
}
```

Quando o sistema operacional aloca memória, ele o faz sempre com blocos de tamanho múltiplos de uma página. Não há como alocar menos que 4 KiB por vez. Se um tamanho de bloco cuja granularidade não seja do tamanho de uma página for requisitado, seu código desperdiçará memória, uma vez que uma página inteira será alocada de qualquer maneira. O que *malloc* e outras rotinas de alocação da *libc* fazem é reaproveitar páginas já alocadas sempre que possível.

Claro que ainda há o problema dos buffers de tradução (TLBs)... É possível que quando o código tente acessar uma página diferente, não exista ainda um TLB válido contendo o cache da tradução do endereço linear. Ou, pior, pode ser que todos os TLBs estejam em uso e que algum tenha que ser invalidado. Neste caso o processador terá que fazer uma “paradinha” para traduzir o endereço e atualizar uma TLB. Ao atualizar uma TLB o processador pode ter que “escrever” o conteúdo do cache associado a ele, piorando a situação...

No caso da arquitetura *Haswell*, os dTLBs ('d' de 'data') poderão realizar cache de tradução de 64 páginas ao mesmo tempo, o que nos dá 256 KiB para páginas de 4 KiB. Mais do que isso e teremos dTLBs inválidos que precisarão ser validados, tornando o acesso à memória mais lento. Ao manter a menor quantidade possível de páginas alocadas, você contribui para o aumento de performance da sua aplicação.

Adicione a limitação de espaço no cache L1d e você verá quanto problema de performance pode conseguir encarando memória como um recurso “sem limites”, como ensinam nos cursos de análise de sistemas...

Ao invés de usarmos malloc, podemos usar a syscall *mmap*, que alocará uma ou mais páginas, alinhadas, para nós. Uma desvantagem de usar *mmap* ao invés de *malloc* é que, para “liberar” o bloco alocado, é necessário usar *munmap* passado o endereço e também o tamanho do bloco previamente alocado (diferente de *free*, já que as estruturas de *malloc* mantém essa informação!).

Eis um exemplo simples de uso de *mmap*:

```
#include <stdio.h>
#include <sys/mman.h>

...
void *ptr;
size_t blksize = 4096*10;

/* Aloca 10 páginas, deixa que o kernel escolha o endereço. */
if ((ptr = mmap(NULL,
                 blksize,
                 PROT_READ | PROT_WRITE,
                 MAP_ANONYMOUS,
                 -1, 0)) == NULL)
{
    /* erro! */
    ...
}

/* Usa o ponteiro 'ptr' aqui... */
...

/* "libera" o bloco. */
munmap(ptr, blksize);
...
```

## Um exemplo de injeção de código, usando páginas

A turma que gosta de exploits vai adorar essa. Usar a função *mmap* para alocar páginas nos dá alguns poderes que *malloc* não tem. Por exemplo, podemos alocar uma página, injetar um código em *linguagem de máquina* nela, desabilitar o bit XD e saltar para o código.

Esse é o princípio do compilador JIT (*Just In Time*). Um código, numa linguagem qualquer, é compilado e o código é colocado em páginas, em *runtime*! Claro que não vou montar um compilador aqui, mas ai vai um exemplo interessante de *code injection*:

```
/* exploit.c */
#include <stdio.h>
#include <memory.h>
#include <sys/mman.h>

/* Tamanho de uma página. */
#define PAGE_SIZE 4096

/* Código que vai ser injetado. */
const unsigned char code[] = {
    0x48, 0x89, 0xf8,    /* mov rax,rdi; */
    0x48, 0x01, 0xf8,    /* add rax,rdi; */
    0xc3                  /* ret; */
};

int main(int argc, char *argv)
{
    long (*fp)(long);
    int x = 3, value;

    /* Aloca uma única página.
       Poderíamos alocar o tamanho suficiente para caber o código, mas
```

```

    mmap vai alocar uma página de qualquer jeito! */
fp = mmap(NULL, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
if (fp != NULL)
{
    /* Copia o código para a página */
    memcpy(fp, code, sizeof(code));

    /* Desabilita a escrita na página e habilita a execução. */
    mprotect(fp, PAGE_SIZE, PROT_READ | PROT_EXEC);

    /* Executa o código injetado via ponteiro. */
    value = fp(x);

    /* Dealoca a página! */
    munmap(fp, PAGE_SIZE);

    printf("O dobro de %d é %d.\n", x, value);
}
else
    puts("Não consegui alocar uma página!");

return 0;
}

```

O código injetado, acima, é bem simples. Ele não contém referências à memória. Se tivesse, teríamos que “ajustá-lo” em *runtime*, tomando como base o endereço base da página e endereços de variáveis ou funções, do programa em C. Esses ajustes têm o nome técnico de *fix-ups*.

Considere esse pequeno código:

```

bits 64
section .text

func:
    call f
    ret
f:
    mov rax,rdi
    add rax,rax
    ret

```

O NASM criará um arquivo objeto assim:

```

00000000 E8 01 00 00 00    call 6
00000005 C3                  ret
00000006 48 89 F8          mov  rax,rdi
00000009 48 01 C0          add  rax,rax
0000000C C3                  ret

```

O que segue o valor 0xE8 é o endereço “relativo” para onde o CALL saltará em relação a próxima instrução. É o valor que será acrescentado ao RIP.

Esse código não precisa de fix-ups, mas se fossemos implementá-los para essa rotina teríamos que manter um ponteiro para a posição depois de 0xE8 e adicionar 5 a esse “unsigned int”, completando o ajuste. Existem casos onde uma versão do CALL usa um endereço absoluto (um endereço linear) e, neste caso, ele terá 8 bytes de tamanho. Neste caso os fix-ups são obrigatórios. É o caso de saltos indiretos.

Criar um *code injection*, desse jeito é trabalhoso... Se você quiser um JIT compiler “real” no seu código, recomendo o uso da *libJIT*<sup>51</sup>.

**Aviso:** Usar a sessão *.data* ou *.bss* para conter um código executável costuma falhar porque o sistema operacional aloca páginas para essas sessões com atributos que impedem a execução de código (bit XD setado). Se você tentar fazer algo assim:

```
char buffer[1] = { '\xc3' }; /* C3 = RET */
```

---

51 Download em <https://www.gnu.org/software/libjit/>

```

/* Declara um ponteiro para uma função e coloca o endereço do buffer nele. */
void (*fptr)(void) = (void (*)(void))buffer;

/* Chama a função via ponteiro. */
fptr();

```

A chamada 'fptr()' vai causar um “segmentation fault”. E a mesma coisa vai acontecer se você alocar um buffer com *malloc*. O que fiz com *mmap* foi criar uma entrada de página com o atributo PROT\_EXEC, ou seja, o bit XD zerado!

## Quanta memória física está disponível?

É importante que sua aplicação tente não usar mais memória do que a fisicamente disponível. Isso causará menos *page faults* e evitará *swapping*, mantendo a performance de sua aplicação previsível. Para tanto, você terá que manter um registro de quanta memória está usando e quanta memória está disponível.

No Linux, você poderá trabalhar com duas funções: *sysinfo* e *getusage*. Com a primeira você obtém informações globais, que concernem ao sistema operacional. Com a segunda você obtém informações relacionadas com o seu processo.

A função *sysinfo* é uma syscall e retorna a seguinte estrutura:

```

struct sysinfo {
    long uptime;          /* Seconds since boot */
    unsigned long loads[3]; /* 1, 5, and 15 minute load averages */
    unsigned long totalram; /* Total usable main memory size */
    unsigned long freeram; /* Available memory size */
    unsigned long sharedram; /* Amount of shared memory */
    unsigned long bufferram; /* Memory used by buffers */
    unsigned long totalswap; /* Total swap space size */
    unsigned long freeswap; /* swap space still available */
    unsigned short procs; /* Number of current processes */
    unsigned long totalhigh; /* Total high memory size */
    unsigned long freehigh; /* Available high memory size */
    unsigned int mem_unit; /* Memory unit size in bytes */
    char _f[20-2*sizeof(long)-sizeof(int)]; /* Padding to 64 bytes */
};

```

Não é coincidência que essa função retorne os mesmos valores que você obtém usando o comando *free*:

	total	used	free	shared	buffers	cached
Mem:	7.7G	1.3G	<b>6.4G</b>	0B	50M	836M
Low:	7.7G	1.3G	6.4G			
High:	0B	0B	0B			
-/+ buffers/cache:		468M	<b>7.2G</b>			
Swap:	7.9G	0B	7.9G			

Acima temos os valores pessimista (em vermelho) e otimista (verde) da memória física livre. Para obtê-los, via *sysinfo*, basta usar os macros:

```

#define MIN_FREE_RAM(si) ((si).freeram)
#define MAX_FREE_RAM(si) ((si).freeram + (si).sharedram + (si).bufferram)

```

Linux tentará liberar a memória usada por buffers e caches quando as aplicações demandarem recursos. Na maioria das vezes é seguro obter a memória física livre com o método otimista, mas nem sempre...

Quanto ao uso de recursos pela sua aplicação, Linux usa um conceito chamado *Resident Set Size* (RSS). Literalmente “tamanho do conjunto residente”. É a memória física usada pelo processo atual ou, se você quiser, incluindo os processos filhos. Assim como *sysinfo*, a função *getusage* retorna uma estrutura contendo diversas estatísticas sobre o seu processo, incluindo a quantidade de page

faults, operações de swap executadas pelo processo e até mesmo chaveamentos de contexto de tarefas. Diferente de *sysinfo*, os valores obtidos são retornados em KiB, não em bytes.

No caso do Windows, você pode usar a função *GetProcessMemoryInfo* para obter os valores de RSS (que, na nomenclatura da Microsoft, é chamado de *Working Set Size*). Essa função é parte da PSAPI (pode ser necessário importar a psapi.dll, dependendo da versão do seu Windows). Para obter a memória física livre do sistema, use *GlobalMemoryStatus*.



# Capítulo 11: Threads!

Você provavelmente tem um processador que possui mais de um **núcleo** e deve estar se perguntando se threads não são a solução definitiva para ganho de performance... Infelizmente, não!

*Threads* existem para “dividir o trabalho”, permitindo a execução de uma mesma rotina em várias frentes, teoricamente, em paralelo. Existe um potencial ganho de performance geral, já que dois ou mais processos trabalhando em partes diferentes podem terminar o trabalho que uma rotina discreta levaria o dobro do tempo... Mas não é sempre assim. É necessário entender como as threads funcionam para tirar boa vantagem delas...

Para entender o que é, de fato, uma thread é preciso entender os conceitos de **contexto de tarefa** e a diferença entre *tarefa* e *thread*. Ambas são unidades de execução e ao falar de ambas, falamos de paralelismo.

“Tarefa” está associada a um recurso de infraestrutura presente no processador, desde o 286, que se refere ao *chaveamento de contexto de tarefa*, que **não é implementado** no modo x86-64. Pelo menos, não da mesma forma que o é no modo i386.

**Contexto de tarefa** é o conjunto de valores de todos os registradores que estavam sendo usados por um processo antes dele ser interrompido. TODOS os registradores, incluindo:

- Os de uso geral: De RAX até R15, RSP, RIP, RBP, RSI e RDI;
- Os seletores de segmento CS, FS e GS (os demais não são usados no modo x86-64);
- Os registradores SIMD (XMM0 até XMM15, e/ou YMM0 até YMM15);
- A pilha do coprocessador matemático;
- RFLAGS

Quando o kernel muda de uma tarefa para outra, ele salva o contexto de tarefa inteiro em algum lugar, carrega o contexto da outra tarefa de outro lugar e salta para o CS:RIP da nova tarefa. Só assim há a garantia de que a nova tarefa continuará exatamente de onde parou. Essa troca de tarefas é chamada de **chaveamento de tarefa** ou *task switching*. Notou que esse chaveamento implica, necessariamente na interrupção da execução de uma tarefa e o reinício de outra?

*Thread*, por outro lado, é um conceito mais abrangente. Em processadores com vários núcleos, ou sistemas com vários processadores, podemos executar tarefas de forma verdadeiramente paralela (chama-se *Simetrical Multi Processing*, ou SMP). Mas uma thread pode também ser executada em fatias de tempo (*Time slicing Multi Processing*) e, neste caso, haverá *task switching*.

No que concerne o chaveamento de tarefas, isso só pode ser feito mediante uma entrega do controle do seu programa, que é executado no *userspace*, para o kernel (*kernelspace*). A forma como isso é feito depende do sistema operacional, mas a maioria dos sistemas decentes o fazem de forma preemptiva...

## Multitarefa preemptiva e cooperativa

Nos idos do início dos anos 90 a multitarefa cooperativa era o *state-of-the-art* dos sistemas operacionais de baixo custo. Reinava o Windows 3.1. Por “cooperação” entende-se que a aplicação entregava o controle ao kernel chamando funções da API e, dentre outras coisas, este aproveitava o ensejo para executar o código de um *scheduler*.

*Scheduler*, numa tradução literal, é um agendador. É uma rotina que decide quanto e quais tarefas

serão chaveadas, dando a impressão de multiprocessamento. No esquema de multitarefa cooperativa a aplicação “cooperava” com o sistema operacional, por assim dizer, entregando o controle de tempos em tempos para o kernel. O problema é que, se na aplicação surgisse um loop infinito ou nenhuma função da API fosse chamada, a ilusão de multiprocessamento era quebrada.

Um dos modos como o Windows fazia isso era através de uma chamada explícita para uma função de gerenciamento da fila de mensagens contida na função *WinMain* da aplicação:

```
...
while (GetMessage(&msg))
{
    TranslateMessage(&msg)
    DispatchMessage(&msg);
}
...
```

A função *GetMessage* executava o scheduler, além de obter a última mensagem da fila...

Já os sistemas UNIX sempre foram baseados em multiprocessamento preemptivo. “Preemptividade” é a capacidade do kernel de interromper uma tarefa sem a cooperação do código no *userspace*. Ao contrário da crença popular, preemptividade não tem nada haver com processamento simultâneo. O significado da palavra é evidente: do dicionário, um dos sinônimos é “antecipado”. Aqui isso quer dizer que o kernel controla o chaveamento, não o seu programa!

## Múltiplos processadores

Todo processador executa código apontado pelo par de registradores CS:RIP. Para fazer isso o processador é colocado num modo específico (x86-64, no nosso caso) e uma série de inicializações são feitas para aproveitar todo o recurso de hardware à disposição da CPU (disco, memória, USB etc). Num ambiente multiprocessado não é muito diferente...

Seja num sistema com múltiplos processadores ou num processador com múltiplos “núcleos”, um deles é “eleito” para ser o *bootstrap processor* (BSP) e este controla o funcionamento de todos os demais, chamados de *application processors* (AP). O BSP é inicializado no modo protegido e paginado pelo sistema operacional e os APs devem ser colocados no mesmo modo, já que todos eles são inicializados, logo depois do *power up*, no modo real. Depois de inicializados, os APs são “colocados para dormir”, através da instrução HLT (Halt) e serão “acordados” pelo BSP quando o sistema operacional precisar executar uma thread de forma simétrica.

Para isso, **todos** os processadores do seu sistema executam parte do kernel. Só que o BSP não pára nunca, nunca é colocado na cama, ninguém dá um beijinho de boa noite e deseja “bons sonhos”. Ele não pode parar, é o chefe! No caso dos APs, se não estiverem executando nada, estarão dormindo. O BSP pode, então, enviar uma **interrupção** para acordá-los, enviando também um endereço lógico, completo, relativo ao CS:RIP da thread que será executada...

Repare que os APs têm seus próprios conjuntos de registradores, suas próprias unidades de execução, seus próprios caches, etc. Os APs podem ser processadores físicos independentes ou processadores lógicos, contidos nos núcleos de sua CPU. Tanto faz! Para o kernel esses processadores são todos isolados, comunicando-se via um barramento de dados isolado chamado ICC (*Interprocessor Commuunication Channel*).

Já que interrupções têm um papel tão importante nesse tipo de ambiente, a Intel resolveu incorporar no chip da CPU um *controlador local programável avançado de interrupções* (“Local APIC” ou LAPIC) em cada um dos processadores lógicos. Antigamente o PIC (*Programmable Interrupt Controller*) era um chip externo à CPU, desde o Pentium o LAPIC já acompanha o chip.

Esses LAPICs contém, além do controle de interrupções, timers e identificação do processador. E ,

através do barramento ICC, o BSP pode enviar interrupções para os APs, iniciando, suspendendo ou interrompendo threads.

O código responsável por decidir para onde uma thread vai ser “enviada”, ou se ela vai ser contextualizada para ser chaveada, é tarefa de um pedaço de código do kernel chamado *scheduler*.

### Como o *scheduler* é chamado?

A maneira mais fácil é através de interrupções ao processador. Certos dispositivos no seu computador enviam um pedido de interrupção ao processador pedindo atenção. Por exemplo, quando você pressiona uma tecla em seu teclado o circuito associado a ele pede que o processador pare tudo o que está fazendo para executar uma rotina específica que lê as portas do teclado e armazene os códigos relativos à tecla pressionada. Quanto essa rotina termina, o processador retorna à execução que foi interrompida.

Uma das maneiras de interromper o processador de forma previsível é usar algum tipo de *timer* para enviar requisições de interrupção ao processador com frequência conhecida. Por exemplo, podemos programar um timer de alta resolução para pedir interrupções a intervalos de 18 ms<sup>52</sup>. Se o scheduler for chamado pela rotina da interrupção desse timer, ele terá chance de chavear tarefas a cada 18 ms de intervalo.

Dessa forma, o scheduler interrompe tarefas e reinicia outras de acordo com um algoritmo complicado que leva diversos fatores em conta: Prioridade da tarefa (quais tarefas terão maiores ou menores fatias de tempo para si), paralelismo real versus time slicing etc. No caso do Linux, o nome adotado para o algoritmo é CFS (*Completely Fair Scheduler*). O que nos interessa saber é que o scheduler realizará chaveamento de tarefas, quando for necessário, e manterá registros de tarefas paralelizadas (via SMP).

### Finalmente, uma explicação de porque SS não é zero no modo x86-64!

Quando falei sobre os seletores no modo x86-64 mostrei que apenas o seletor CS é considerado pelo processador. Todos os outros (DS, ES, FS, GS e SS) são ignorados. Na ocasião, mostrei um pequeno código para imprimir o conteúdo dos seletores e, para surpresa geral, o registrador SS contém um valor diferente de zero e, pior, com um RPL condizente com o *userspace*!

Existe apenas um motivo, pelo que posso perceber: Quando é feito um chaveamento de tarefa entre o *ring 0* e o *ring 3*, via interrupção, o processador salva o conteúdo de SS:RSP na pilha e **zera** SS. O sistema operacional pode, então, manter um valor válido em SS no *userspace* como uma maneira de saber, rapidamente, em que *ring* ele está. Mais importante: SS poderá conter um índice para um descritor que tenha informações importantes para o kernel, sobre o processo que foi chaveado!

### Na prática, o que é uma “thread”?

Para simplificar, uma thread é uma função. Essa função é executada como se estivesse num processo separado, mas compartilhando todos os recursos do processo que criou a thread. De fato, threads também são conhecidas como *lightweight processes*, ou “processos leves”. Além do nome, não há muita diferença entre *threads* e *processos*: Ambos têm pilha e contextos próprios, por exemplo.

Sobre “recursos compartilhados com o processo”, quero dizer que suas threads enxergam todas as variáveis globais do seu programa. Isso é diferente de um *fork*, onde um novo processo é criado

---

<sup>52</sup> Consulte a configuração do seu kernel via “sudo sysctl kernel.sched\_latency\_ns”. No meu caso, a latência do scheduler é de 18 milisegundos (ou 18000000 de nanosegundos).

com base na cópia do processo original e os dados do novo processo só são “copiados” se forem modificados pelo processo filho (*copy-on-write*). Isso não acontece com threads.

Um detalhe sobre a pilha assinalada a uma thread: É aconselhável que ela seja pequena para não colocar pressão no sistema de paginação. Além do possível chaveamento de tarefas, o uso de muitas páginas não compartilhadas (pilhas separadas) pode exaurir a capacidade do cache L1d e das TLBs, causando grandes atrasos.

Outra coisa importante é saber que não é aconselhável disparar um grande número de threads (ou processos). Lembre-se que seu processador tem número limitado de núcleos (e, em sistemas multiprocessados, um número limitado de processadores)... Se o número de threads simultâneas suplantar o número de “processadores lógicos”, o kernel terá, necessariamente, que realizar chaveamento de tarefas, ao invés de usar processamento simétrico (SMP).

Num ambiente Linux, por exemplo, é possível termos até 32768 processos simultâneos (veja */proc/sys/kernel/pid\_max*), incluindo ai todas as threads além daquelas associadas diretamente aos processos. Isso é um exagero, é claro! Esse número de threads, num processador capaz de lidar com 8 threads em SMP (é o caso do i7) significa que teremos 4096 chaveamentos de contexto, que deixarão cada uma das tarefas bem lenta... Considere o caso em que um chaveamento de tarefa pode ser feito a cada 18 ms e que 32768 tarefas tenham o mesmo nível de privilégio. Com 4096 chaveamentos, teremos latência de 73 segundos entre as tarefas. Ou seja, uma tarefa executa 18 ms de processamento e é colocada para dormir durante 73 segundos (4096 chaveamentos vezes 18 ms) até que seja “acordada” novamente!

Nesse cenário o seu código cuidadosamente desenhado para atingir a melhor performance possível será tão lento que te dará vontade de bater com a cabeça na parede, com força!

## Criando sua própria thread usando pthreads

Pthreads ou *Posix Threads* é a biblioteca padrão, em ambientes POSIX (obviamente), para lidar com threads (mais obvio ainda!). Trata-se de um conjunto de funções para criar e manusear threads. E é bem fácil de ser usado.

Resumidamente, quando você cria sua thread está criando uma ramificação paralela do fluxo de execução, desassociado do fluxo da thread principal (a thread do processo). Assim, teremos dois fluxos de execução. Em algum momento teremos que “juntar” (*join*) o fluxo novo com o fluxo da thread principal. Essa “junção” coloca a thread principal para dormir, enquanto ela espera pelo término da thread secundária.

Usar threads com pthreads é essencialmente algo assim (omiti todos os tratamentos de erro para facilitar a leitura):

```
pthread_t tid;           /* Identificador da nova thread. */
pthread_attr_t tattr;    /* Atributos da nova thread. */
void *mythread(void *); /* Protótipo da função que será executada na nova thread. */
void *param_ptr;         /* Ponteiro para os parâmetros que a
                           thread receberá. */
int retval;              /* Valor retornado pela thread, se algum.
                           Pode ser de qualquer tipo. Uso 'int' como exemplo apenas. */

/* Inicializa os atributos default da thread, alterando
   apenas o tamanho da pilha para o menor possível. */
pthread_attr_init(&tattr);
pthread_attr_setstacksize(&tattr, PTHREAD_STACK_MIN);

/* Cria e põe a nova thread em execução. */
pthread_create(&tid, &tattr, mythread, param_ptr);
```

```

/* Continua a fazer algo na thread principal enquanto a thread secundária 'roda'. */
...
/* Espera pela hora de 'juntar' a thread criada com a principal.
   Coloca a thread principal para dormir enquanto a thread identificada por 'tid' não
   retorna. */
pthread_join(tid, &retval);
/* Neste ponto a thread secundária já não existe mais. */

```

Note que antes de criarmos a thread temos que informar seus atributos. A função *pthread\_attr\_init* inicializa esses atributos com valores default. Eis alguns deles:

- A thread será criada como **joinable**;
- O nível de prioridade é 0 (default);
- A pilha da nova thread tem tamanho default.

Pode ser necessário usar uma função do tipo *pthread\_attr\_setXXX* (onde XXX é o atributo a ser modificado) para fazer ajustes finos. No exemplo acima, um dos atributos que modifiquei foi o tamanho da pilha da thread. Por default, *pthread\_create* criará pilhas do mesmo tamanho informado por 'ulimit':

```
$ ulimit -s
8192
```

Como podem ver, no meu sistema o tamanho de pilha default é de 8 MiB<sup>53</sup>. A função *pthread\_attr\_setstack* precisa receber um valor igual ou superior à constante PTHREAD\_STACK\_MIN, caso contrário, a thread não será criada (*pthread\_create* retornará um valor diferente de zero, indicando erro).

Quanto ao atributo *joinable*, é perfeitamente possível alterá-lo para criar threads desassociadas da thread principal. Assim, não precisaremos 'juntá-las'. Isso exige que você tenha o controle da vida da thread, isto é, se ela ainda está em execução ou não.

Você também pode alterar o nível de prioridade da thread. Níveis maiores que zero tendem a dar mais *time slices* para a thread do que para aquelas com privilégio menores. A função *pthread\_attr\_setschedparam* pode ser usada para isso.

Outro atributo que você pode achar interessante é a *afinidade* da thread. Você pode dizer em qual processador lógico quer que a thread seja executada usando a função *pthread\_attr\_setaffinity\_np*. Mas recomendo que esse recurso seja deixado a cargo da inicialização default e, em última análise, do próprio sistema operacional... O sufixo 'np' no nome da função significa “non portable”. É uma dica para que você evite usar funções assim...

## Criando threads no Windows

Usar a API do Windows para criar threads é também bem fácil, mas o controle das threads não é tão facilitado assim. Não há a facilidade de “juntar” as threads, por exemplo. Diferente de *threads* as threads do Windows são desassociadas da thread principal. Mas existe uma maneira de emular um “join”, como pode ser visto mais adiante...

Para criar uma thread só precisamos usar a função *CreateThread*, que toma seis parâmetros:

---

<sup>53</sup> Não que a thread ou o processo criem uma pilha de 8 MiB. O sistema aloca uma página “presente” (4 KiB) e as demais páginas como “não presentes”. Assim, pelo processo de *page fault* pode-se fazer a pilha crescer até 8 MiB, se necessário.

```

HANDLE WINAPI CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadSecurityAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreatingFlags,
    LPDWORD lpThreadId
);

```

Na API do Windows, geralmente os valores default são assumidos ao se passar um ponteiro NULL (ou zero). É o caso dos dois primeiros parâmetros e dos flags de criação (atributos?). Eis um exemplo de criação de uma thread:

```

HANDLE hthread;

/* Protótipo da nossa função que será "paralelizada". */
DWORD WINAPI MyThreadFunc(LPVOID);

/* "cria thread" e coloca para executar. */
if ((hthread = CreateThread(NULL,
                            128*1024,           /* Pilha de 128 KiB. */
                            MyThreadFunc,
                            NULL,
                            0,
                            NULL)) == NULL)
{
    ... trata erro aqui ...
}

... faz alguma coisa ...

/* "join" */
WaitForSingleObject(hthread, INFINITE);

```

Se o parâmetro do tamanho da pilha for 0 então *CreateThread* usará uma pilha de 1 MiB de tamanho (esse é o tamanho de pilha default do Windows).

O parâmetro do tipo **LPTHREAD\_START\_ROUTINE** é um ponteiro para uma função executada pela thread secundária, do mesmo tipo usada na chamada à *pthread\_create*. Há também o ponteiro genérico para o parâmetro que será passado para essa função, os atributos da thread (*dwCreateFlags*) e o ponteiro para o identificador da thread.

Diferente de *pthread\_create*, podemos criar uma thread suspensa setando o bit **CREATE\_SUSPENDED** nos flags de criação. Criar uma thread suspensa, em pthreads, não é possível. Mas, há meios de simular esse comportamento. Uma dessas maneiras é trabalhar com sinais. Outra, usando artefatos de sincronização como mutexes.

Outra diferença entre Windows API e pthreads é o primeiro parâmetro de *CreateThread*. Já que a infraestrutura de segurança do Windows é meio esquisita, é necessário informar se o identificador da nova thread pode ser ou não compartilhado por processos filhos. Normalmente passamos NULL nesse parâmetro, deixando o Windows escolher a melhor característica de segurança para a nova thread.

Assim como em pthreads, se sua thread não foi criada suspensa ela é colocada em execução imediatamente. Caso contrário você terá que chamar *ResumeThread*, passando o identificador. Se sua thread está em execução e, através de uma outra thread, você queira colocá-la em estado suspenso, basta executar *SuspendThread*, passando o identificador.

Já disse que a metáfora de “juntar” threads não existe na API do Windows. Você pode esperar pelo término de uma thread via chamada à função *WaitForSingleObject*. Essa função, em essência, funciona do mesmo jeito que *pthread\_join*, exceto que você não obtém o código de retorno da thread através dela. Para obter esse valor de retorno terá que usar a função *GetExitCodeThread*.

Existe outra função de espera: *WaitForMultipleObjects*, que tem uma vantagem sobre as múltiplas

chamadas a *pthread\_join* necessárias para “juntar” múltiplas threads secundárias... Se tivermos que esperar por 3 threads, por exemplo, podemos fazer:

```
HANDLE tids[3];
int trys;

/* Coloca os handles das threads no array. */
tids[0] = tid1;
tids[1] = tid2;
tids[2] = tid3;

/* Tenta esperar pelo fim, de todas as threads, 10 vezes.
   Espera 3 segundos (3000 milissegundos) a cada vez. */
trys = 10;
while (--trys &&
       WaitForMultipleObjects(3, tids, TRUE, 3000) == WAIT_TIMEOUT)
{
    /* Sinaliza, de alguma maneira, para as threads que elas precisam terminar! */
}

if (trys <= 0)
{
    ... trata erro aqui ...
}
```

## **Parar uma thread “na marra” quase sempre não é uma boa ideia**

Em ambos os casos, tanto no Windows quanto no POSIX, é possível parar uma thread “na marra”. No caso do Windows temos a função *TerminateThread*, no caso do UNIX, *pthread\_kill*. Elas não fazem a mesma coisa: *TerminateThread* força o término de uma thread de forma que ela não tem opção de continuar o processamento até algum ponto de controle. Isso pode causar sérias instabilidades no seu código, já que você poderá estar esperando que algum estado esteja presente, durante a execução de seus programas que não será atualizado devido ao término “inesperado” da thread.

No caso das pthreads, a função *pthread\_kill* envia um sinal especificado à thread. Esta pode ter a chance de tratá-lo (a não ser que o sinal seja SIGKILL ou outro sinal não tratável, de acordo com a especificação POSIX).

Pthreads também implementa a função *pthread\_cancel*, que é um apelido para *pthread\_kill* enviando um sinal SIGUSR1 ou SIGUSR2 para a thread a ser cancelada. Se for matar uma thread “na marra”, prefira usar *pthread\_cancel*.

De qualquer maneira, parar uma thread de forma “não graciosa” é sempre uma má ideia. É imprescindível que seus códigos criem e destruam threads de uma maneira estável. Evite sempre “matar” uma thread sem mais nem menos.

## **Nem todas as threads podem ser “mortas”**

Para que *pthread\_cancel* funcione, se você realmente precisar matá-la na marra, é preciso que existam “pontos de cancelamento”. Como regra geral, toda função que use recursos do sistema operacional são pontos de cancelamento. Uma thread que está em loop infinito vazio, por exemplo:

```
for (;;);
```

Pode permanecer em loop infinito eternamente, mesmo que um *pthread\_cancel* seja usado.

A técnica mais usada, nesse caso, é usar uma *syscall*, por exemplo, a função *sleep*, pedindo que o processo “durma” por zero segundos:

```
for (;;) sleep(0);
```

Toda syscall é um ponto de cancelamento. Isso dá a chance de podemos cancelar a thread. Tem ainda o efeito benéfico de fazer com que o scheduler não morra de fome. Se não há chamadas para o kernel, via *syscall*, então o scheduler só é chamado via interrupção, causando um pico de consumo de processamento.

É claro que uma chamada a *sleep* adicionará ciclos ao seu processo. O que é mais um argumento contra o aumento de performance automático assumido para processos multithreaded.

Esse exemplo do loop infinito é meio pobre. Existem meios de colocarmos uma thread para dormir sem recorrermos a uma *syscall*. Por exemplo, podemos usar *pthread\_sigwait* para esperar que um sinal seja enviado para a thread.

Em assembly existe a instrução PAUSE, que age como se fosse um NOP, exceto que oferece uma dica ao processador de que a thread atual encontra-se em loop. PAUSE não substitui uma chamada via *syscall*, mas é útil nas rotinas de sincronismo, como *spin locks*. O motivo da existência de PAUSE nada tem haver com o scheduler, é claro. O processador tem lá seus problemas com muti processamento também!

## **Threads, núcleos e caches**

Um dos grandes problemas com as threads, e ambientes com múltiplos processadores ou núcleos, é que os estados dos caches serão compartilhados em algum momento (já que threads criadas no seu programa compartilham o mesmo espaço de endereçamento). É bom lembrar que o cache L1 existe separadamente para cada processador lógico, ou seja, para cada núcleo de seu processador, mas podemos estar usando algum sistema com mais de um processador também. Assim, duas threads podem ser atrasadas pelo protocolo de consistência de caches dos processadores (ou núcleos).

Considere duas threads (A e B) executando de maneira simétrica. Se a thread A escreve numa variável global 'x' e a thread B lê a mesma variável, temos duas threads acessando a mesma região de memória e, como consequência disso, alguma coisa tem que ser feita para manter a consistência dos caches. Consistência, aqui, significa que todos os processadores verão a mesma coisa nos caches sempre que possível. Não é interessante que a cópia do cache L1 de um processador seja feita para os demais. Isso só é possível graças a um protocolo de consistência chamado MOESI (Modified, Owned, Exclusive, Shared, Invalid), cuja descrição está além do escopo deste texto.

O motivo de eu te falar deste protocolo é deixar claro que threads tem o potencial de causar problemas com caches, especialmente caches L1d, o que pode adicionar muitos ciclos de máquina de perda de performance nos seus programas. De novo, threads não são o recurso barato para alcançar performance elevada...

## **Trabalhar com threads não é tão simples quanto parece**

Como os dados e código de threads são compartilhados, o que acontece é que as várias threads que podem estar associadas a um único processo pai podem acessar a mesma variável. Isso tem o potencial de gerar alguns absurdos...

Pode parecer estranho, já que uma instrução normalmente é completada antes de uma interrupção, mas considere o caso do incremento de uma variável do tipo 'long', no modo i386: Neste modo não podemos usar os registradores estendidos ou atualizar valores na memória que tenham tamanho maior que 32 bits (a não ser que usemos MMX ou SSE). O incremento de uma variável 'x', do tipo 'long', seria mais ou menos assim:

```
add  dword [x],1  
adc  dword [x+4],0
```

Se houver um chaveamento de tarefa entre as instruções ADD e ADC, o incremento de “x” será feito pela metade! E esse não é o único caso esquisito. Num cenário semelhante, suponha que uma thread incremente uma variável inteira e outra thread escreva ‘0’ na mesma variável. O que acontece?

O algoritmo da primeira thread está esperando um valor incrementado (maior que zero?), mas quando a tarefa for chaveada de volta para ela, a variável conterá zero, colocado lá pela outra thread!

Sempre que uma das threads quer modificar uma variável usada por outra thread pode acontecer esse monte de problemas que são chamados de ***race conditions***.

## ***Evitando “race conditions”***

Para evitar esses problemas usa-se o artifício de “sincronizar” as threads. Trata-se de uma espécie de sinal de trânsito, um *semáforo*, usado para temporariamente interromper o processamento de uma ou mais threads enquanto uma delas detém o controle sobre o recurso compartilhado. É como se esse semáforo ficasse verde para uma thread e vermelho para as outras. Quando uma thread obtém um sinal vermelho ela entra num estado de espera. É colocada para “dormir” até o sinal ficar verde<sup>54</sup>.

Quem diz para o semáforo ficar vermelho são as próprias threads. Quem acionar o botão que pede atenção, primeiro, passa a ser a dona do semáforo. Ou, melhor, quem obtém a “trava” (*lock*) primeiro, não permite que as outras threads o travem até que o semáforo seja destravado (*unlock*).

Por exemplo, se quisermos alterar uma variável global em nossas threads, devemos fazer algo assim:

```
pthread_mutex_lock(&mtx);
if (++x < MAX_X)
    dosomething();
pthread_mutex_unlock(&mtx);
```

Mutexes<sup>55</sup> são um tipo de objeto de sincronismo de threads. Eles são traváveis e destraváveis pela thread que os controla... Se já estiver travado, então a rotina entra em loop e coloca a thread para dormir. O código entre o *lock* e o *unlock* é chamado de ***critical section***. Não é à toa: Essa é a parte crítica da thread que precisa ser sincronizada...

A sincronização deve ser usada com cautela. Se você travar toda a sua thread então não há sentido em criar threads! O sincronismo deve ser feito em pedaços chave do código da thread (as sessões críticas) para evitar os tipos de conflito causados por *race conditions*, mas é só... Trata-se de ter o poder de parar outras threads semelhantes, temporária mente, para que um pedaço do processamento sempre seja feito de forma correta, sem a “interferência da concorrência”.

Existem tipos de objetos de sincronização diferentes que funcionam como “trava”. Além dos mutexes temos *spinlocks* e *semaphores*. Essencialmente, eles são a mesma coisa, mas cada um têm lá seus conjuntos de vantagens e desvantagens. A explicação sobre cada um deles está além do escopo deste livro.

Embora o sincronismo resolva, em parte, as *race conditions*, pode causar outro: ***dead locks***. Acontece quando duas threads obtém travas (*locks*) de objetos de sincronismo diferentes de forma que nenhuma das duas threads seja capaz de destravá-los individualmente, deixando ambas em estado de espera eterno. Em certas circunstâncias *dead locks* podem ocorrer porque tanto no

---

<sup>54</sup> Semáforos têm um sentido específico com relação aos esquemas de sincronização. Estou usando aqui a analogia, não o objeto *semaphore*.

<sup>55</sup> Mutex é uma abreviação de “MUTually Exclusive”.

processamento simétrico quanto no chaveamento de tarefas não temos ideia da posição em que nossas threads se encontram, do ponto de vista da execução... O sincronismo pode ser feito numa sequência que você não considerava possível, deixando uma ou as duas travas eternamente “travadas”.

Um cenário alegórico: Você e um amigo têm chaves para abrir duas travas de uma porta de um quarto... Vocês entram no quarto e fecham as travas, mas seu amigo morre e leva a chave dele consigo (caiu num poço no meio do quarto, por exemplo)... Você jamais sairá do quarto porque a fechadura que só abre com a chave do seu amigo está morta pra você (*dead locked*).

## Threads e bibliotecas

Nem toda função pode ser usada impunemente numa thread. Para que possa, ela tem que ser reentrante, ou seja, não usar recursos externos à função (como variáveis globais, por exemplo). Não é o caso de muitas funções da *libc*, por exemplo.

Para o Linux, a *libc* é bem documentada pela *Free Software Foundation*. A documentação completa pode ser obtida no link <http://www.gnu.org/software/libc/manual/>. E, por lá, você encontrará, para qualquer função, algumas dicas sobre o uso com ambientes multithreaded... Toda função tem uma dica, depois do protótipo:

```
size_t strlen (const char *s)
[Function]
Preliminary: | MT-Safe | AS-Safe | AC-Safe
The strlen function returns the length of the null-terminated string s in bytes. (In other words, it returns the offset of the terminating null character within the array.) ...
```

Este “MT-Safe” nos diz que podemos usar *strlen* em códigos *multithreaded* sem medo. Mas, atenção! O fato de uma função ser *thread safe* não significa que ela seja atômica, quer dizer, que ela seja executada totalmente antes que um chaveamento de contexto seja feito. Também não significa que não possamos ter problemas com *race conditions*. Considere o caso da função *strcpy*, que tipicamente é implementada como:

```
char *strcpy(char *dest, const char *src)
{
    char *p = dest;
    while (*dest++ = *src++);
    return p;
}
```

Se duas threads usarem o mesmo ponteiro de destino para as chamadas a *strcpy*, provavelmente teremos uma *race condition*, mesmo que *strcpy* não use nenhuma variável global ou “estática local”.

Assim, os problemas que listei antes continuam valendo, mesmo para as funções que são “ok” para serem usadas em threads.

As dicas “AS-Safe” e “AC-Safe” referem-se a sinais. No caso de AS-Safe, a função pode ser usada em rotinas que respondem as “interrupções” causadas por sinais. No caso de “AC-Safe” a função é um ponto de cancelamento de thread.

Um exemplo de função da *libc* que é thread *unsafe* é *strtok*. Ela é explicitamente marcada como “MT-Unsafe”. Ela mantém um estado global para que a segunda chamada a ela funcione como deve. Existe uma versão thread safe: *strtok\_r*, que toma um parâmetro adicional onde o programador deve armazenar o estado da função.

## **Threads e Bloqueios**

Além de *thread safety* existe a questão de bloqueios de I/O. Algumas funções precisam ser completadas antes que possam ser chamadas novamente. Funções como *write* são *syscalls* que precisam ser completadas. No caso desta função ser chamada por duas threads, a segunda thread a chamá-la parecerá estar “dormindo”, enquanto a primeira está executando a função.

Algumas syscalls podem ser configuradas para serem “non blocking”, mas isso só significa que a função retornará imediatamente com algum código de erro, enquanto executa a requisição em background. É o caso de funções de *sockets*. O fato de você configurar o descriptor de arquivo criado pela função *socket* como “non blocking”, não significa que seus pacotes serão dispatchados em paralelo! Significa que eles serão serializados pelo kernel e colocados numa fila e a função que o fez irá te dizer isso (e você precisa tratar essa informação!).

Lembre-se que o comportamento padrão para a maioria das syscalls é “blocking”.

## **O que significa isso tudo?**

É claro que multiprocessamento, quando implementado com cuidado, tem o potencial de acelerar alguns processos, mas tenha em mente que a quantidade de cuidados é grande e exige estudo intenso do que a thread deve fazer. Por causa da complexidade, o potencial para atrapalhar a boa performance de suas rotinas é muito grande.

O que temos até agora? *Task Switching* grava e recupera o contexto de tarefas. Esses contextos são chaveados através de software contido num *scheduler*, que é executado, provavelmente, pela rotina de resposta de interrupções de timers atendidas pelo processador. Cada tarefa pode ser executada em regime de *time slicing*, onde uma fatia de tempo é dada, pelo scheduler, para cada tarefa ou de forma simétrica, distribuídas entre os processadores lógicos. Temos ainda a necessidade de sincronização de threads por causa de recursos compartilhados e a necessidade de se precaver contra *dead locks*...

Se isso não fosse o suficiente, muitos problemas com paginação e caches podem surgir, atrasando ainda mais a CPU. No que o multiprocessamento é útil, afinal de contas?

Em grande parte, os sistemas operacionais modernos usam multiprocessamento para lidar com a percepção do usuário. Se duas tarefas parecem estar sendo feitas ao mesmo tempo, o usuário fica feliz. Mas, no contexto de performance, provavelmente as tarefas estão sendo feitas mais lentamente do que se fossem feitas isoladamente.

Mesmo com essa má notícia, o multiprocessamento pode ser muito útil, em certos casos. Por exemplo, para quebrar as iterações de um loop em várias pequenas iterações individualizadas em threads. Esse tipo de “divisão de tarefas”, se for feito via SMP, tem mesmo o potencial de acelerar a rotina. É o que bibliotecas como *OpenMP* tentam fazer...

## **Tentando evitar o chaveamento de contextos de tarefas**

Num cenário ideal teríamos apenas uma thread sendo executada em cada núcleo o processador lógico, evitando o *task switching*. Em teoria, isso faria com que nossos processos usem o total poder de processamento do processador. No entanto, outros processos também estão em execução, o que estraga a coisa toda.

Tudo o que podemos fazer é tentar minimizar a quantidade de threads que colocaremos em execução. A primeira coisa que precisamos saber para chegarmos a esse objetivo é: “Quantos processadores eu tenho?”. Eis uma rotina que nos devolve essa informação para vários sistemas operacionais:

```

/* threads.c */
#ifndef __linux__
#include <unistd.h>
#elif defined(__WINNT__)
#include <windows.h>
#elif defined(__FreeBSD__)
#include <sys/param.h>
#include <sys/sysctl.h>
#endif

static int get_number_of_cores(void);

int get_number_of_processors(void)
{
#if defined(__linux__)
    return sysconf(_SC_NPROCESSORS_ONLN);
#elif defined(__WINNT__)
    /* FIXME: Essa função só existe no Win7 e Win2008-R2 ou superiores. */
    return GetMaximumProcessorCount(ALL_PROCESSOR_GROUPS);
#elif defined(__FreeBSD__)
    int num_processors, r[2];
    size_t size;

    /* Pega `sysctl hw.ncpu` */
    r[0] = CTL_HW;
    r[1] = HW_NCPU;
    if (sysctl(r, 2, &num_processors, &size, NULL) == -1)
        return get_number_of_cores();
    return num_processors;
#else
    return get_number_of_cores();
#endif
}

static int get_number_of_cores(void)
{
    int cores;

    __asm__ __volatile__ (
        "movl $1,%eax\n"
        "cpuid\n"
        : "=b" (cores)
    );

    return (cores >> 16) & 0xff
}

```

## Usando o OpenMP

Alguém percebeu que algumas rotinas podem aproveitar o paralelismo, disponível nos processadores modernos *automaticamente*. Um padrão foi divulgado para permitir que isso fosse feito sem que o desenvolvedor altere muito os códigos pré-existentes. Trata-se da biblioteca *OpenMP* (MP, de *MultiProcess*).

Suponha que você queira preencher um array de 400 inteiros com zeros. A função óbvia para fazê-lo seria algo assim:

```

for (i = 0; i < 400; i++)
    array[i] = 0;

```

Mas, e se pudessemos dizer ao compilador que esse loop pode ser dividido em loops menores distribuídos entre os núcleos do processador? Isso é possível graças ao OpenMP e à diretiva do compilador *pragma*:

```

#pragma omp parallel for
for (i = 0; i < 400; i++)
    array[i] = 0;

```

Usar essa diretiva diz ao compilador para injetar código que quebrará o loop em *n* loops individuais,

onde  $n$  é o número de processadores (que o OpenMP sabe, sozinho, qual é!). É como se tivéssemos um *fork* para  $n$  processos e depois um *join*, ao final do loop.

Crie uma função simples, coloque o fragmento de código acima (incluindo o header *omp.h*) e crie a listagem em assembly... Ao dar uma olhada na listagem você verá um código enorme, que faz uso de funções da *libgomp* (GNU OpenMP).

Essencialmente, usar *OpenMP* é uma questão de usar um pragma que é aplicável para um bloco:

```
#pragma omp parallel [tipo] [cláusulas]
{
    ... código que será 'paralelizado' aqui ...
}
```

Os modificadores 'tipo' e 'cláusulas' são opcionais e servem para otimização e para evitar alguns problemas. No nosso exemplo usamos o tipo 'for' para otimizar o paralelismo para loops. Mas, não usamos quaisquer cláusulas. O exemplo abaixo mostra a necessidade de uma cláusula:

```
#pragma omp parallel for private(j)
for (i = 0; i < 400; i++)
    for (j = 0; j < 100; j++)
        matrix[i][j] = 0;
```

Sem a cláusula *private*, acima, provavelmente o compilador tentaria paralelizar os dois loops. Ao dizer que a variável 'j' é privada ao bloco, dizemos que o loop interno não deve ser paralelizado. Isto é, só o loop externo será dividido em threads.

Podemos ser mais explícitos e adicionar a cláusula 'shared', com a lista de variáveis compartilhadas pelas threads:

```
#pragma omp parallel for shared(i) private(j)
...
```

Existem outras diretivas além de 'parallel' e outras cláusulas além de 'shared' e 'private' na especificação.

## ***OpenMP* não é mágico**

Só um lembrete: Já que estamos falando de paralelismo, *race conditions* também podem aparecer no uso de *OpenMP*. A biblioteca também fornece recursos de sincronização para tentar minimizar o problema, o que também pode acarretar em *dead locks*... Em resumo, os mesmos problemas que você teria com threads “manualmente” construídas em seu código, pode obter com *OpenMP*.

## **Compilando e usando OpenMP**

Além de usar pragmas, temos que dizer ao compilador que estamos usando *OpenMP* e, quando formos linkar, usar a biblioteca *libgomp* (no caso do GCC). Se precisarmos usar funções da especificação do *OpenMP* em nossos códigos, temos que incluir o header *omp.h* também. Uma compilação com OpenMP ficaria assim:

```
$ gcc -O3 -march=native -fopenmp -c -o test.o test.c
$ gcc -o test test.o -lgomp
```

Sem usarmos '-fopenmp' e linkarmos com a *libgomp*, todos os pragmas serão inóquos.

**Atenção:** OpenMP não substitui *pthread*s. E também não é uma panacéia! Ele existe como tentativa de aproveitar o paralelismo *simétrico* de algumas arquiteturas. De fato, se você chamar a função *omp\_get\_max\_threads*, da *libgomp*, verá que ele devolve apenas o número de processadores lógicos contidos no seu sistema. A diferença é que via *pthread*s você pode criar tantas threads quanto o

sistema operacional permitir, mas com OpenMP você só pode criar um número limitado delas... É claro, não há garantias de que as threads criadas pela libgomp serão simétricas, mas essa é a ideia...

## ***OpenCL e nVidia CUDA***

O “CL” em OpenCL significa *Concurrency Library* e CUDA é “Compute Unified Device Arquitecture”. A primeira (OpenCL) é uma biblioteca genérica, usada para abstrair as unidades de processamento em seu sistema, a segunda é proprietária da nVidia e serve para usar as unidades de processamento da sua placa de vídeo (GPU).

No caso do OpenCL o programador aloca “dispositivos” de processamento, sem necessariamente saber quais são eles. Daí, faz o “upload” de pequenos programas chamados “kernels” (não confundir com o kernel do sistema operacional) que são executados em paralelo nesses dispositivos.

Numa máquina *standalone* (seu computador caseiro) o OpenCL usará os seus processadores lógicos e os diversos “cores” da GPU contida na sua placa de vídeo. Tome como exemplo a placa de vídeo que posso em minha estação de trabalho: Uma nVidia GT-635. Segundo o site da nVidia essa placa tem 384 “CUDA cores”. Meu processador é um i7, com mais 8. Com OpenCL posso, em teoria, criar programas que executem 392 threads concorrentes.

Num ambiente mais profissional, podemos mapear processadores remotos para juntarem-se ao time... Imagine um *data center* com 100 máquinas iguais a minha. Podemos ter o valor teórico de 39200 threads concorrentes executando, mesmo em máquinas diferentes.

Na prática, usar tantas threads causam muitos problemas, especialmente com relação à sincronização. A linguagem usada pelo OpenCL (e pelo CUDA) ajuda um bocado, já que é parecida com C, mas tem algumas características próprias, dedicadas a esse tipo de ambiente.

Não vou mostrar um exemplo aqui. Se quiser saciar sua curiosidade procure documentação sobre OpenCL e saiba que os drivers de sua placa de vídeo disponibilizam bibliotecas dinâmicas para que você possa usar esse recurso.

# Capítulo 12: Ponto flutuante

Neste capítulo quero mostrar que lidar com ponto flutuante não é uma coisa tão simples quanto parece. Existe muita confusão sobre o assunto porque estamos acostumados a lidar com esses tipos de valores, com componentes fracionárias em base decimal, e pensar neles como “exatos”. Não é o que um computador faz. Ele sempre lida com valores em formato binário e inteiros. Ponto flutuante é um artifício, incorporado nos processadores atuais, que permite **representar** valores fracionários.

Essa representação não é perfeita. É aproximada. Entender os problemas causados por essas imperfeições te farão evitar ou minimizar erros de “imprecisão” ao usar tipos como *float* e *double*.

## Precisão versus Exatidão

No nosso dia a dia, quando se fala em precisão estamos falando de o quanto próximo da realidade um valor calculado pode estar de um valor exato. Isso implica em algum tipo de arredondamento. Eis um exemplo simples: O valor 2 é exato, mas  $\frac{1}{3}$  não é. A fração  $\frac{1}{3}$  é composta de dois valores exatos, mas seu resultado só pode ser expresso com alguma aproximação.

O termo precisão precisa ser melhor definido: No contexto da aritmética com ponto flutuante ele significa a quantidade de dígitos ou algarismos usados na representação do valor. Ele **não** significa a quantidade de dígitos ou algarismos “depois da vírgula”. É uma distinção importante e deve ser mantida em mente para o melhor entendimento do restante deste capítulo.

## O que é “ponto flutuante”?

Todos os tipos de dados, em linguagens de programação como C, têm tamanho fixo. Assim como *char*, *int* e *long*, os tipos *float*, *double* e *long double* não são exceções. O tipo *float* tem exatamente 32 bits de tamanho, o *double* tem 64 e o *long double*, 80. Para facilitar a discussão sobre “ponto flutuante” lidarei apenas com o tipo *float* daqui por diante. Pode-se extrapolar a explicação para *double* e *long double* aumentando o tamanho de seus componentes, como veremos à seguir.

Quanto maior o tipo, maior é a quantidade que pode ser armazenada em seu interior. Veremos mais adiante que *float* pode representar valores com precisão de, pelo menos, 7 algarismos decimais. Para conseguir usar mais algarismos temos que recorrer a tipos “maiores” como *double* e *long double*.

Se podemos usar apenas 7 algarismos num tipo *float*, valores como 3,1415926 não podem ser representados de forma “exata”. Graças à limitação da precisão esse valor transforma-se em 3,141593. Note que ele tem que ser arredondado para que a precisão de 7 algarismos seja atendida.

Usando a precisão de 7 algarismos, se multiplicarmos 3,141593 por 10 a vírgula flutuará para a direita, obtendo 31,41593. Continuamos com a precisão de 7 algarismos, mesmo que tenhamos uma quantidade inferior de “casas depois da vírgula”. Essa “flutuação” da vírgula (ou do ponto) é o que dá nome a esse tipo de aritmética.

## Estrutura de um *float*

Como todo tipo usado em linguagens de programação, a atribuição da faixa de valores válidos, em base decimal, é uma ilusão conveniente. Os tipos lidam somente com valores binários. A fórmula abaixo nos diz como um valor do tipo *float* é armazenado<sup>56</sup>:

56 Para o tipo *double* basta considerar 'e' com 11 bits e a mantissa 'm' com 53. O valor 127 torna-se 1023.

$$f = (-1)^s \cdot 1.m \cdot 2^{e-127}$$

Os valores “s”, “m”<sup>57</sup> e “e” (iniciais de “sinal”, “mantissa” e “expoente”) são obtidos da estrutura binária do tipo:

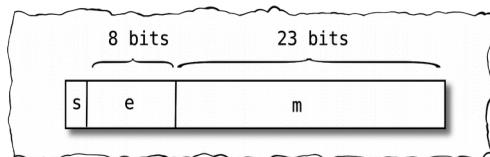


Figura 15: Representação de um float.

A mantissa corresponde a parte fracionária do valor em ponto flutuante. Embora a palavra “mantissa” não seja uma boa definição para a parte fracionária de um número real, já que é definida, matematicamente, como “parte fracionária” de um “logaritmo”. Na falta de um termo melhor, e por motivos históricos, usarei esse termo mesmo.

Quando digo que a mantissa é somente a parte fracionária, quero dizer que a parte inteira do valor armazenado na estrutura é implícito e sempre igual a 1, em binário. Abaixo, falo sobre a analogia com a “notação científica”, que explica melhor o porquê do uso dessa técnica. No caso de um *float* a mantissa tem 23 bits de tamanho.

Isso significa que a quantidade de bits no campo *m* tem, realmente 24 bits de tamanho. Embora a mantissa seja a parte fracionária o que temos é um valor inteiro, positivo, de 24 bits que, quando multiplicado por  $2^e$  representará um número que contém uma parte fracionária.

No float o expoente *e* tem 8 bits e varia de 0 a 255, fazendo com que a potência de dois, usada para escalar o valor binário (ou “deslocar a vírgula”), na fórmula acima, possa variar entre -127 e 128. No restante do capítulo falarei de *e* como se ele tivesse sinal e usarei um *E*, maiúsculo, para obter o expoente real. Tenha em mente que o sinal é obtido a partir da subtração de *e* com o máximo número positivo de 8 bits. Ou seja, o expoente de base 2 é sempre calculado como:

$$E = e - 127$$

Da estrutura sobra apenas o bit *s*. Se esse bit estiver setado, significa que o valor representado na mantissa é negativo, caso contrário, positivo.

O campo *m* é a parte interessante. Ele é inteiro, mas representa valores fracionários, já que seus bits são ordenados a partir da posição -1 (bit mais significativo) até a posição -23 (bit menos significativo). Desconsiderando o expoente *e* (fazendo igual a 127 e, portanto *E* = 0), pode parecer que o menor valor positivo que pode ser armazenado na mantissa é 1 (o bit implícito) e o próximo valor possível seria  $1 + 2^{-23}$ . Veremos mais adiante que esse degrau é chamado de  $\epsilon$  (letra grega “epsilon”) e é importante para podermos fazer comparações entre valores... Eu disse “pode parecer” porque há outra maneira de codificar um valor na estrutura de um *float*.

### Analogia com “notação científica”

O motivo de adicionarmos 1 antes do *m* pode ser entendido, por analogia, ao conceito de **notação científica**. Em física, é costumeiro expressar valores muito grandes ou muito pequenos usando potências de 10 (ou seja,  $10^x$ ) e limitando a parte inteira do valor a apenas um algarismo diferente de zero. Para escrever o valor 10 bilhões é mais fácil escrever  $1,0 \cdot 10^{10}$  do que 10000000000. Gasta menos espaço.

Num valor em ponto flutuante acontece a mesma coisa. O único valor diferente de zero que a

<sup>57</sup> A mantissa *m* é representada em binário nessa notação. E o expoente *e* em decimal. Isso pode confundir, mas é a maneira que tenho para compreender o que acontece... Pense em *e* como o deslocamento do “ponto binário”, ao invés de uma potência de base 2...

porção inteira pode assumir é 1 porque, em notação científica na base 2, o único algarismo diferente de zero que temos é 1. Ele é implícito para poupar espaço, ou melhor, para acrescentar um bit na exatidão do valor, e não precisa estar codificado na estrutura do *float*. Assim, num tipo *float*, temos sempre 24 bits de precisão: O bit 1, implícito, e os 23 bits da mantissa.

Note que qualquer número dentro da faixa de precisão de um *float* pode ser expresso dessa maneira. Por exemplo, 0.5 pode ser escrito, em binário, como  $+0b1.00000000000000000000000 \cdot 2^{-1}$ . Isso será codificado num *float* como  $s=0$ ,  $e=126$  (para  $E=-1$ ) e  $m=0$ . O valor inteiro contido numa variável *float* será 0x3F000000, como mostro no exemplo abaixo:

```
/* test.c */
#include <stdio.h>

void main(void)
{
    float x = 0.5f;

    printf("%#08X", *(unsigned int *)&x);
}
-----%<---- corte aqui -----%<-----
$ gcc -o test test
$ ./test
0x3F000000
```

### **Valores especiais na estrutura de um *float*.**

Existem quatro exceções importantes ao esquema previamente apresentado: O valor 0.0 é um dos casos especiais. Se, na estrutura do *float*, o  $e$  for zero e a mantissa também, então teremos um valor zero.

A segunda exceção acontece quando  $e$  for zero e a mantissa não for. Neste caso temos um valor em ponto flutuante **denormalizado** ou **subnormalizado**. Essa categoria de valores recebe esse nome porque não segue à norma (que assume um valor inteiro 1.0 implícito). Esses valores especiais são usados para representar valores bem próximos de zero e seguem a fórmula:

$$f = (-1)^s \cdot 0.m \cdot 2^{-126}$$

As outras duas exceções acontecem quando  $e$  for igual a 255 (0xff). Neste caso, se  $m$  for zero temos a representação de “infinito”, mas se  $m$  for diferente de zero temos a representação de algo que **não é um número** (*NaN* ou *Not A Number*). Valor *NaN* ocorre quando, por exemplo, tentamos extrair a raiz quadrada de números negativos ou quanto tentamos calcular 0/0.

Existem dois tipos de *NANs*: *qNaNs* e *sNaNs*. O *q* usado como prefixo na sigla significa “quieto”. Em casos como a extração de raiz quadrada de valores negativos, o valor obtido é um *qNaN*. A diferença entra um *qNaN* e um *sNaN*, sendo esse último um *NaN* “sinalizado”, é que este último representa um erro... Não que um *qNaN* também não seja, mas consumir um *sNaN* pode ser usado para disparar uma rotina de tratamento de erros enquanto um *qNaN* é facilmente ignorado.

Eis algumas operações que resultam em *NaN*:

Operação	Motivo
$0/0$	Valor indeterminado.
$0 \cdot \pm\infty$	Valor indeterminado.
$+\infty + -\infty$ , $-\infty + +\infty$	Valores indeterminados.
$\tan \frac{\pi}{2}$ , $\tan \frac{3\pi}{2}$	Tangente desses arcos não existem.
$\log_{10} -n$ , $\log_2 -n$ , $\log_e -n$	Logarítmos de valores negativos não existem.
arc sen, arc cos	Arco-seno e arco-cosseno de valores fora da faixa entre $[-1, +1]$ são inválidos.
Qualquer operação envolvendo NaNs.	NaN “não é um valor”!

Tabela 9: Casos onde ocorrem NaNs

Operações como  $0^0$  e  $1^\infty$  são definidas como tendo resultado igual a 1,0 e, portanto, não resultam em NaNs, mas para valores diferentes de 0 e  $\pm 1$  a potência  $n^{+\infty}$  provavelmente resultará em NaN.

### Pra que os valores denormalizados existem?

Pode parecer que  $2^{-126}$ , que é o menor número normalizado que pode ser representado num *float*, seja pequeno o suficiente para qualquer possível valor que queiramos usar, mas acontece que em operações aritméticas podem ocorrer *underflows*.

*Underflow*, é claro, é o contrário de *overflow*. Um *overflow* acontece quando extrapolamos o máximo valor possível e *underflow*, o mínimo... Os valores denormalizados são a maneira que o padrão IEEE 754 achou de criar *underflows progressivos*. Ou seja, se o resultado de uma operação for denormalizado ele ainda é válido, mas o desenvolvedor deve tomar cuidado...

Abaixo de  $2^{-126}$ , o menor valor que pode ser armazenado num número denormalizado é  $2^{-149}$ . É claro que a faixa aumenta se estivermos falando de tipos como *double* e os *underflows* ficarão ainda menores.

### Intervalos entre valores

Por causa do número limitado de bits não há como representar todos os valores no domínio dos números reais. No caso de *float*, o menor incremento entre valores que pode ser obtido (desconsiderando o escalonamento) é da ordem de  $2^{-23}$ , ou seja, se o LSB da mantissa estiver setado. Se a mantissa tem 23 bits de tamanho e o bit mais significativo corresponde ao valor  $2^{-1}$ , então o bit menos significativo dela corresponde ao valor  $2^{-23}$ .

Esse valor ( $2^{-23}$ ), onde desconsideramos o escalonamento ( $E=0$ ), é chamado de *epsilon* e é representado pela letra grega  $\epsilon$  e pela constante `FLT_EPSILON`, no header *float.h*. Ele é definido como o menor incremento possível para o valor 1,0. Se o valor representado no *float* for maior que 1,0, então  $\epsilon$  precisará ser escalonado de acordo, para obtermos o menor degrau de variação possível

Com os valores denormalizados o bit mais à direita que representaria  $\epsilon$  estará escalonado por um fator de  $2^{-126}$ , ou seja, o menor degrau de um valor denormalizado é de  $2^{-103}$ . Mas a faixa dos valores denormalizados é pequena em relação a toda a faixa dos valores normalizados. Como regra geral podemos considerar que o valor de  $\epsilon$ , para intervalos entre 1 e 2, é ele próprio ( $2^{-23}$ ), para valores entre 2 e 4 o degrau aumenta para  $2 \cdot \epsilon$ , já que o último bit desaparecerá (o “ponto” foi deslocado para a direita!), entre 4 e 8 o valor aumenta para  $4 \cdot \epsilon$  e assim por diante.

Da mesma maneira, entre 1 e  $\frac{1}{2}$  o valor de  $\epsilon$  será de  $\frac{1}{2} \cdot \epsilon$ , entre  $\frac{1}{2}$  e  $\frac{1}{4}$  ele muda para  $\frac{1}{4} \cdot \epsilon$  e assim por

diante.

Conforme os intervalos vão ficando maiores devido ao incremento do expoente, maior vai ficando o intervalo entre os valores “representáveis”. A figura abaixo ilustra:

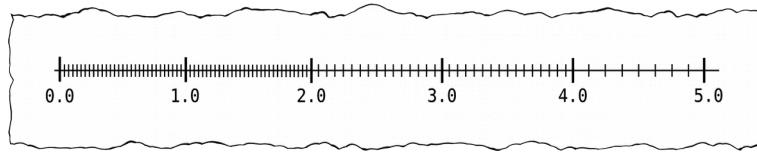


Figura 16: O intervalo mínimo entre valores (*epsilon*).

Repare que a cada vez que ganhamos um bit na parte inteira, perdemos um na parte fracionária, por isso o intervalo dobra.

Para valores muito grandes, por exemplo,  $1 \cdot 10^7$ , a menor distância entre valores adjacentes será de aproximadamente 1,0. Lembre-se que, num *float*, a precisão é de, aproximadamente, 7 algarismos decimais e, como 10 milhões tem 7 algarismos, a menor distância será de 1... Se o valor fosse  $1 \cdot 10^{10}$ , a menor distância entre valores adjacentes será da ordem de 1000.

Esses são cálculos aproximados que são melhor compreendidos em binário, usando a estrutura do *float*... O valor  $1 \cdot 10^7$  será codificado como  $s=0$ ,  $e=160$ ,  $m=0x1502F9$ . O valor de  $e$  desloca a “vírgula” para a direita em 33 bits ( $E = 160 - 127 = 33$ ) e o valor de  $m$ , adicionando o valor 1.0, implícito, torna-se  $0b1.00101010000001011111001$ . Deslocando a “vírgula” teremos  $0x2540BE400$  que dá, exatamente, 10000000000 em decimal. Se adicionarmos 1 ao LSB da mantissa teremos  $m=0x1502FA$  que, ao somarmos 1.0 e deslocarmos a vírgula obteremos  $0x2540BE800$ , que dá exatamente 10000001024, em decimal.

Isso torna os cálculos de valores monetários de grande monta um problema. O que acontece se um sujeito ou uma empresa tem R\$ 10 milhões em alguma conta e há um rendimento de R\$ 500,00? O resultado final continuará sendo R\$ 10 milhões e os R\$ 500,00 que foram adicionados será perdido, já que **não pode** ser armazenado num *float*! O intervalo entre valores adjacentes devem ser sempre levados em consideração com base no maior valor usado na operação!

## Um problema de base

Assim como na nossa batida base 10, em base 2 existem valores que não podem ser expressos exatamente. Na base 10 as frações  $\frac{1}{3}, \frac{1}{7}, \frac{1}{11}, \frac{1}{13}$  etc não podem ser expressas exatamente. Você obterá uma dízima periódica. Se usássemos outra base numérica, como a base 3, por exemplo, a fração  $\frac{1}{3}$  seria exatamente igual a 0,1<sub>(3)</sub>. Mas, o processador trabalha com a base 2...

Eis outra surpresa: Em relação aos valores fracionários, só é possível escrever com exatidão os valores terminados em 0 e 5, na base 2! (0,5, por exemplo é exatamente  $2^{-1}$ ). Para entender porquê, eis um programinha que nos mostra a estrutura de *floats*:

```
#include <stdio.h>

/* 0.4, 0.6, 0.8 e 0.9 são múltiplos de 0.2 e 0.3. Não os coloquei aqui para não
   usar muito espaço em mostrar os resultados. */
float values[] = { 0.1, 0.2, 0.3, 0.5, 0.7 };
#define ARRAY_SIZE (sizeof(values) / sizeof(values[0]))

void main(void)
{
    int i;

    for (i = 0; i < ARRAY_SIZE; i++)
        printf("%.1f: 0x%08X\n", values[i], *((unsigned *)&values[i]));
}
-----%----- corte aqui -----%-----
$ gcc -o imprecise imprecise.c
```

```
$ ./imprecise
0.1: 0x3DCCCCCD
0.2: 0x3E4CCCCD
0.3: 0x3E99999A
0.5: 0x3F000000
0.7: 0x3F333333
```

Dá para perceber que, em todos os casos, exceto com 0.5, o valor da mantissa é uma dízima periódica? O compilador só resolveu arredondar alguns valores “para cima” para que esses fiquem mais próximos do valor “real”.

Tomemos 0,1 como exemplo: Em decimal ele só pode ser transformado em  $0b1.100110011001\dots \cdot 2^{-4}$ . Faça as contas e verá que algo semelhante ocorre com os outros valores que citei... A dízima periódica só pode indicar uma coisa: Aproximação. Nunca exatidão!

Vendo de outra forma, cada bit da mantissa equivale a uma potência de 2 com expoente negativo. Depois do 1,0 implícito temos 0,5; 0,25; 0,125; 0,0625; 0,03125; ... Os valores sempre terminam em 5! Na verdade eles terminam em 1, em binário... Assim, na hora de arredondar, o compilador somará um 5 na última casa decimal.

É como aquela regrinha de arredondamento que usávamos na escola...

## O conceito de valor “significativo”

Quantas vezes, durante o período escolar, você não teve que fazer cálculos usando o valor  $\pi$  com duas casas decimais apenas? Ao invés de usar 3,141592653589793238462643383279502 você não usou 3,14? Dependendo da grandeza do que você teve que calcular o valor 3,14 atendia muito bem. Daí a precisão de 2 “casas depois da vírgula” era significativa e você ignorava todo o resto (afinal,  $\pi$  é um valor irracional!). A mesma coisa acontecia com o valor da aceleração da gravidade  $g$ ...

Na verdade você não “ignorava” todo o resto, você somava 0,005 e **depois** ignorava todas as “cadas decimais” além da segunda.

Outro exemplo: Quando vai calcular distância percorrida por um trem naqueles famigerados probleminhas de Física, desconsiderava grandezas inferiores a metros, por exemplo... Centímetros ou milímetros, neste caso, não tem lá grande influência no que significa “distância”, no resultado do problema.

A biblioteca padrão faz a mesma coisa com os valores em ponto-flutuante. Algumas imprecisões são simplesmente ignoradas, após o arredondamento, quando imprimimos valores via `printf()`, por exemplo. Um erro de  $1 \cdot 10^{-8}$ , devidamente escalonado em relação ao expoente, é jogado fora. Não tem “significado” no cálculo!

Isso não significa que o processador não use esse pequeno erro para obter o valor final. É na hora de mostrá-lo que o erro é desconsiderado através de arredondamento. Este é um detalhe muito importante ao lidarmos com ponto flutuante: Os cálculos são feitos sempre com os valores como eles estão e é na hora de mostrá-los que arredondamentos são feitos!

Mas, arredondamentos não são tão simples assim. Existem quatro tipos diferentes:

- “Para cima”, em direção ao  $+\infty$ ;
- “Para baixo”, em direção ao  $-\infty$ ;
- Em direção ao zero e;
- O mais próximo possível.

Normalmente o processador é configurando para arredondar para o valor “mais próximo possível” e existem diversas vantagens nisso do ponto de vista matemático. Os outros tipos de arredondamento

podem ser usados em casos especiais, bastando reconfigurar a unidade aritmética.

O arredondamento resolve o problema da limitação de bits da mantissa, mas gera outros interessantes...

## Regras da matemática elementar nem sempre são válidas

Além da impossibilidade de obter um valor real a partir de uma divisão por zero ou da extração de uma raiz quadrada de um número negativo, você também deve ter aprendido na escola que as operações fundamentais têm as seguintes propriedades:

- **Comutativa:**  $a + b = b + a$  ou  $ab = ba$ ;
- **Associativa:**  $a + (b + c) = (a + b) + c$  ou  $a(bc) = (ab)c$
- **Distributiva:**  $a(b + c) = ab + ac$

O que nos força a aceitar um fato surpreendente sobre aritmética com ponto flutuante: Das três regras, apenas a comutativa é sempre válida. As propriedades associativa e distributiva nem sempre ocorrem. E não estou falando de valores de grandezas diferentes. Eis um exemplo:

```
double x = 0.1 + (0.2 + 0.3);
double y = (0.1 + 0.2) + 0.3;

/* Imprimirá "x é diferente de y"!
if (x == y)
    puts("x é igual a y");
else
    puts("x é diferente de y");
```

O problema ocorre porque, graças ao arredondamento, existem erros diferentes para cada uma das representações de valores acima. O valor 0.2, por exemplo, pode ter um erro de arredondamento um pouquinho maior do que a representação do valor 0.1 e 0.3, por exemplo. Assim, o valor 0.5, calculado a partir dos valores aproximados de 0.2 e 0.3, não é exato, é arredondado.

O que quero dizer é que você não está vendo adições exatas no exemplo ai de cima. O 0,3 calculado a partir da adição de 0,1 e 0,2 não é o mesmo 0,3 que é adicionado em seguida, no caso da variável  $y$ . Você obterá um valor um pouco fora do 0,6 esperado. A mesma coisa acontece com o cálculo da variável  $x$ , mas com um erro um pouco diferente do 0,6 calculado para  $y$ ... Assim, os dois 0,6 serão diferentes!

Lembra-se que o valor de  $\epsilon$  é maior quando o valor estiver entre 0,5 e 1,0 do que quando ele está entre 0,25 e 0,5 e vai ficando menor quando a faixa vai sendo escalonada por um fator de  $\frac{1}{2}$ ? Ao somar 0,2 e 0,3 obtemos 0,5 somando a um  $\epsilon$  “grande”. E ao somar 0,1 e 0,2 obtemos um 0,3 com um  $\epsilon$  menor que o outro. Quero dizer:

$$\begin{aligned} x &= 0,1 + 0,5 \cdot (1 + \epsilon_1) \\ y &= 0,3 \cdot (1 + \epsilon_2) + 0,3 \end{aligned} \quad \left| \begin{array}{l} \epsilon_1 > \epsilon_2 \end{array} \right.$$

Ao somar os valores restantes teremos a adição de erros parecidos (já que supostamente teríamos os mesmos resultados, mas observe o que obtemos):

$$\begin{aligned} x &= 0,6 \cdot (1 + \epsilon_1) \cdot (1 + \epsilon_1) = 0,6(1 + 2\epsilon_1 + \epsilon_1^2) \\ y &= 0,6 \cdot (1 + \epsilon_2) \cdot (1 + \epsilon_1) = 0,6(1 + \epsilon_1 + \epsilon_2 + \epsilon_1 \cdot \epsilon_2) \end{aligned} \quad \left| \begin{array}{l} \epsilon_1 > \epsilon_2 \end{array} \right.$$

O que obviamente fará  $x$  e  $y$  serem diferentes!

Essas acumulações de erros também ocorrem em multiplicações e divisões. Daí a propriedade distributiva também falha.

Veremos, mais adiante, como lidar com comparações de valores em ponto flutuante, levando esses pequenos erros em consideração.

## **Que tal trabalhar na base decimal, ao invés da binária?**

Desde a especificação ISO C99 (pelo que me lembro), a linguagem C possui mais 3 tipos de “ponto flutuante” além dos *float*, *double* e *long double*. Trata-se de tipos de ponto flutuante decimais: *\_Decimal32*, *\_Decimal64* e *\_Decimal128*. Esses três novos tipos estão na especificação IEEE 754 desde 2008.

O número que segue o tipo diz o tamanho do mesmo, em bits. Mas, sua representação interna é equivalente à decimal, não a binária. Assim, o valor armazenado é, mais ou menos, codificado como:

$$f = (-1)^s \cdot d.m \cdot 10^{e-max}$$

Onde 'd' é um dígito entre 0 e 9, 'm' é a mantissa (em decimal) e o resto é parecido com os formatos de ponto flutuante binário, mas em base 10.

Isso permite a representação daqueles valores: 0,1; 0,2; ...

Mas, existem alguns problemas ao lidar com esses tipos: O primeiro é que eles são pouco performáticos. Há dependência de uso de funções especiais para lidar com eles. O processador não os suporta nativamente. E a codificação dos valores é um tanto complexa.

O segundo problema é que funções da *libc*, como *printf*, não têm suporte a esses tipos. Pelo menos não a *libc* pré-compilada pelo mantenedor das distribuições mais famosas do Linux... Uma maneira de verificar se a *libc* suporta esses tipos é perguntando ao compilador:

```
$ gcc -v 2>&1 | grep "\-\-enable\-\decimal\-\float"
```

Provavelmente o *grep* não encontrará essa string na lista de flags usados na configuração do *gcc*<sup>58</sup>.

O fato de *printf* não suportar os tipos *\_DecimalXX* não significa que eles não possam ser usados. Para mostrá-los teremos que convertê-los para *float* ou *double* e usar a função *printf*. A conversão é feita através de funções *built-in* do compilador. Consulte a documentação do GCC...

Outra coisa: Os valores literais têm um sufixo especial. Por exemplo:

```
_Decimal32 d32 = 1.2df; /* df = "decimal float"? */
.Decimal64 d64 = 0.4dd; /* dd = "decimal double"? */
.Decimal128 d128 = 0.3dl; /* dl = "decimal long double"? */
```

Lembre-se também que esses tipos não resolvem o problema da precisão. Números como 2/3 continuam sendo irracionais...

Minha opinião sobre os tipos ponto flutuante decimais é que eles são inúteis por causa do problema da baixa performance, em relação aos tipos em ponto flutuante binários.

## **A precisão decimal de um tipo *float* ou *double***

Até agora falamos da precisão de um *float* sob o ponto de vista binário. Para calcular a precisão decimal, grosseira, basta usar um logaritmo:

$$p_{decimal} = 1 + \lceil p_{binaria} \cdot \log 2 \rceil$$

Onde *p<sub>binaria</sub>* é a quantidade de bits na mantissa mais o 1.0 implícito, no caso de valores normalizados. E *p<sub>decimal</sub>* é o número de algarismos significativos na base 10. Para o tipo *float* temos

<sup>58</sup> Os caracteres '-', no grep foram “escapados” para que ele não confunda a regular expression com um parâmetro.

$p_{decimal}=8$ . Para o tipo `double`,  $p_{decimal}=17$ .

A especificação de C nos diz que a precisão do tipo `float` é de, pelo menos, 6 dígitos. E do tipo `double`, 15. Isso é condizente com a fórmula acima.

Note que, mesmo que tenhamos um valor do tipo  $1,234567 \cdot 10^{33}$ , isso não é a mesma coisa que um valor inteiro de 34 algarismos (deslocando a vírgula 33 vezes para a direita!). O número de algarismos significativos (a precisão!) não pode ser maior que 7 porque tantos algarismos não podem ser obtidos com a precisão de 24 bits da mantissa de um `float`!

## Comparando valores em ponto flutuante

Graças aos pequenos arredondamentos nos cálculos com valores em ponto flutuante precisamos de um jeito de comparar dois valores “próximos” como se fossem iguais. Fazer uma comparação do tipo abaixo, como vimos antes, pode ser problemática:

```
if (x == y) ...
```

Se  $x$  e  $y$  forem semelhantes, diferindo em um erro pequeno, então elas não serão iguais. Por exemplo, ao comparar uma constante 0,1 com um valor calculado que dê 0,1 como resultado de uma operação aritmética, teremos pequenos erros que tornam ambos diferentes entre si. E a comparação acima vai falhar ( $x == y$  será falso).

Uma maneira melhor de garantir que dois valores próximos sejam considerados como iguais é usando um fator de diferença mínima. O valor de  $\epsilon$  (epsilon):

$$|x - y| \leq \epsilon$$

Se a diferença entre  $x$  e  $y$  for menor ou igual a  $\epsilon$ , então podemos considerá-los iguais. O motivo de termos que comparar o valor absoluto da diferença com  $\epsilon$  é porque tanto  $x$  quanto  $y$  podem ser negativos. Isso nos dá um macro:

```
#define equal(a,b) (fabs((a)-(b)) <= FLT_EPSILON)
```

Mas existe um problema...

Lembre-se que o valor de  $\epsilon$  será escalonado por  $2^n$  quando  $n \neq 0$ . O valor de  $\epsilon$  pode ser pequeno demais para a maioria das comparações! Uma solução é obter o valor de  $\epsilon$  relativo ao maior valor dos dois sendo comparados:

$$\frac{|x - y|}{\max(|x|, |y|)} \leq \epsilon$$

O macro para o cálculo desse “erro relativo” ficaria assim:

```
inline int float_cmp(float a, float b)
{
    float A = fabs(a), B = fabs(b);
    float diff = A - B;

    return diff <= (FLT_EPSILON * max(A, B));
}
```

A especificação ISO C99 nos dá os seguintes macros para comparar valores em ponto flutuante (definidos em `math.h`): `isgreater()`, `isgreaterequal()`, `isless()`, `islessequal()` e `islessgreater()`. O último compara pela diferença. Estranhamente ele não fornece um `isequal()`.

Usar  $\epsilon$  para decidir como comparar valores parece ser uma boa idéia, especialmente quando usamos o erro relativo. Ao subtrair dois valores muito parecidos o resíduo poderá resultar num valor denormalizado ou, pelo menos, bem próximo de zero. Isso **não** significa que você não possa usar

seu próprio  $\epsilon$ . Aliás, é recomendável que o faça! Por exemplo, se minhas contas não precisam ter mais que 4 algarismos depois da vírgula, posso definir um  $\epsilon$  de 0,0001 e criar meus macros substituindo a constante FLT\_EPSILON por uma chamada FLOAT\_EPISOLON, por exemplo, contendo esse valor.

## Não compare tipos de ponto flutuante diferentes

Quando você lida com constantes literais, em C, está atrelando a elas um tipo definido de tipo. Se escreve “1.1” este tipo é, por default, *double*. Se escreve “1.1f” ou “1.1F”, o tipo é *float*.

Comparar 1.1 com 1.1f pode ser desastroso. As precisões são diferentes e, portanto, os erros de arredondamento são diferentes. No código abaixo:

```
float x = 1.1;  
  
if (x != 1.1)  
    printf("São diferentes!");
```

Provavelmente obterá a string “são diferentes!” impressa. O detalhe é que o compilador converterá o valor *double* 1.1 para o tipo *float*, fazendo os arredondamentos necessários e o atribuirá a *x*, **em tempo de compilação**. Mas, no ‘if’ a conversão será feita **em runtime** e no sentido contrário... Pe acordo com a especificação da linguagem, C sempre converte tipos conflitantes para aqueles de maior precisão. Antes de comparar *x* com 1.1 a variável será convertida para *double* (em *runtime*), e usando as regras de arredondamento do processador, não do compilador!

Ou seja, você está comparando laranjas com maçãs...

Regra geral: Mantenha seus tipos, em ponto flutuante, sob controle... Se tiver floats, compare-os com floats... Se forem doubles, compare-os com doubles.

Algumas maneiras de conseguir isso:

- Se quiser usar floats, use o sufixo ‘f’ nos valores literais. SEMPRE!
- Use *type casting*, se não puder usar o sufixo.

O fragmento de código acima poderia ser escrito assim:

```
float x = 1.1f  
  
if (x != 1.1f)  
    ...
```

## Evite overflows!

Uma dica importante é sobre *overflows*. Especialmente se você quer usar ponto flutuante para calcular potências. O exemplo clássico é o do famoso *teorema de Pitágoras*:

$$c = \sqrt{a^2 + b^2}$$

A equação parece inofensiva, mas se um dos valores de *a* ou *b* estourarem, por causa da elevação da potência ao quadrado, você terá um valor inválido dentro da raiz quadrada e o resultado, provavelmente, será um *NaN*. Como evitar isso?

No caso de *Pitágoras*, o método tradicional é elevar ao quadrado apenas o menor valor, retirando o maior valor da raiz quadrada, assim:

$$c = a\sqrt{1 + \left(\frac{b}{a}\right)^2}$$

Se  $(\frac{b}{a})$  não causa um *underflow* então a equação não criará um *overflow* de forma alguma... E, só para satisfazer sua curiosidade, essa nova interpretação do teorema pode ser entendida assim:

$$\sqrt{a^2 + b^2} = \sqrt{\frac{a^2}{a^2}(a^2 + b^2)} = \sqrt{a^2 \left( \frac{a^2}{a^2} + \frac{b^2}{a^2} \right)} = a \sqrt{1 + \left( \frac{b^2}{a^2} \right)}$$

É importante lembrar que os valores denormalizados existem para garantir um *underflow* gradual, mas para *overflows* isso não existe. Assim, é possível que  $(\frac{b}{a})^2$  seja computado com um valor denormalizado, se  $a$  for muito maior que  $b$ .

Neste caso, também é importante escolher o maior dos dois valores para retirar da raiz. A rotina poderia ficar assim:

```
#define swapf(a,b) { float t; t = (a); (a) = (b); (b) = t; }

float hipotenuse(float a, float b)
{
    if (b > a)
        swap(a, b);

    if (!a)
        return b;

    return a * sqrtf(1.0f + (b*b)/(a*a));
}
```

## Ponto fixo

Se você precisa lidar com valores monetários e não quer gastar muito tempo analisando os efeitos de acumulação erros em ponto flutuante, sempre pode usar a técnica do ponto fixo. Diferente do ponto flutuante, essa técnica permite que operações fundamentais sejam feitas em **inteiros** como se contivessem valores em ponto flutuante.

Aritmética com inteiros é bem mais veloz e mais simples. No caso de valores monetários precisamos apenas de duas “casas depois da vírgula” e da parte inteira. Nenhuma das operações precisa de “precisão” maior que essa. Abaixo temos algumas rotinas para demonstrar a técnica:

```
#include <math.h>

#define MUL100(x) ((x) * 100)
#define DIV100(x) ((x) / 100)

/* Rotinas de conversão */
long float_to_money(float x) { return (long)floorf(x * 100.0f); }
float money_to_float(long x) { return (float)x / 100.0f; }

/* Operações elementares */
// long money_add(long x, long y) { return x + y; }
// long money_sub(long x, long y) { return x - y; }
long money_mul(long x, long y) { return DIV100(x * y); }
long money_div(long x, long y) { return MUL100(x) / y; }
```

Com essa técnica todo valor em ponto flutuante é multiplicado por 100. Um valor 1,5 torna-se 150. Um valor 3.1415 torna-se 314. Se você quiser um arredondamento “para cima” basta substituir a função de conversão *float\_to\_money* por:

```
long float_to_money2(float x) { return (long)ceilf(x * 100.0f); }
```

Adições e subtrações podem ser feitas diretamente ou, se preferir, via funções *money\_add* e *money\_sub* (comentadas). O detalhe todo está nas multiplicação e divisão: Já que multiplicamos o valor original, digamos ‘x’, por 100, o que temos é  $(x \cdot 10^2)$ . Ao multiplicar x por y, temos  $(x \cdot 10^2) \cdot (y \cdot 10^2)$  ou então  $xy(10^2 \cdot 10^2)$ . Fica claro que temos uma multiplicação por 100

adicional nessa operação, daí a necessidade de dividir o valor final por 100.

Com a divisão o problema é semelhante... Ao dividir dois valores escalonados por 100, acabamos sem escalonamento algum. Afinal  $\frac{10^2}{10^2}$  nos dá  $10^0$ . Daí a necessidade de multiplicar o dividendo por 100 antes de realizar a divisão.

Para usar as funções basta fazer algo assim:

```
long x, y, r;  
  
x = float_to_money(10.0f);  
y = float_to_money(33.2f);  
r = money_mul(x, y);  
printf("10.00 * 33.20 = %.2f\n",  
      money_to_float(r));
```

As operações serão feitas com inteiros, que são exatos e as únicas conversões ocorrem para obtenção dos valores. E já que estamos lidando com valores exatos, comparações também não causam problemas...

Outra vantagem: O tipo *long* suporta armazenar valores até  $\pm 9 \cdot 10^{18}$ . Multiplicando por 100 limitamos esse valor para  $\pm 9 \cdot 10^{16}$ . Isso é um 9 seguido de 16 zeros, ou 90 quintilhões com duas “casas decimais”. Suficiente para qualquer aplicação que lida com dinheiro, não?

## Modo x86-64 e ponto flutuante

Quando falei sobre misturar C com assembly, falei sobre a convenção de chamada e mostrei que os valores em ponto flutuante são passados através dos registradores XMM0 até XMM7. Isso quer dizer que a arquitetura x86-64 usa SSE. Acontece que ponto flutuante óbviamente existia antes dessa arquitetura e um co-processador matemático (incorporado na arquitetura Intel desde os 486) era necessário.

A forma como o co-processador matemático funciona é um pouco diferente do SSE. Ele possui uma pilha de 8 níveis onde os valores são inseridos e depois das operações feitas. Da mesma forma que é feito numa calculadora HP... Isso chama-se *notação polonêsa reversa* e é um esquema mais complicado e menos flexível do que o do SSE.

O único motivo que você tem para aprender sobre o co-processador matemático é que ele permite precisões ainda maiores que o SSE... Enquanto *float* tem 32 bits de tamanho e *double*, 64. O co-processador suporta valores com 80 bits de tamanho!

## O tipo *long double*

O padrão IEEE 754, que dita as regras para os números em ponto flutuante binários, especifica quatro tipos, em sua última revisão (2008): *single precision*, *double precision*, *extended precision* e *half precision*.

A precisão estendida é conhecida por *long double* e foi adotada pela IEEE **depois** que a Intel incorporou em seu co-processador matemático 8087 o armazenamento de 80 bits. Esse não é um tipo lá muito amigável, mas a mantissa tem 64 bits (contra os 54 do tipo *double* e dos 24 do tipo *float*) e o expoente é maior também (15 bits). O algarismo 1, implícito nos tipos *float* e *double* é explícito nesse formato e é o formato preferido do 8087.

O grande problema do tipo *long double* é que só podemos usá-lo através da pilha do 8087. Isso gera muitas instruções e cria um problema ainda maior: Ao mudar a precisão entre tipos, ocorre o mesmo problema que citei nas operações aritméticas fundamentais: adição de erros de arredondamento!

Como o 8087 realiza todas as operações com precisão máxima, ao adicionarmos dois *floats*, por

exemplo, eles terão que ser, necessariamente, convertidos para *long double* pelo processador e, para isso, precisarão ser arredondados **antes** da operação ser feita. Daí temos triplo arredondamento: O arredondamento de cada operando transformado para *long double*, seguido do arredondamento da operação e, por fim, o arredondamento do resultado de volta ao tipo original.

Isso **não** ocorre com SSE. Bem... ocorrerá se você tentar transformar um *float* num *double* ou vice-versa, mas a carga de *floats* e *doubles* é direta, se você usa SSE ou SSE2.

Operações usando SSE, mesmo no modo i386, tendem a ser mais precisas do que quando usamos o 8087 (o default, no modo i386)... Felizmente, no modo x86-64, SSE e SSE2 são os defaults.



# Capítulo 13: Instruções Estendidas

Em 1996 a Intel anunciou uma extensão para a sua linha de processadores chamada MMX. A coisa causou um alvoroço porque MMX é uma sigla proprietária que significa *MultiMedia eXtension*. A ideia era que os novos processadores lidariam com “multimedia” (ou seja, vídeos!) por hardware. Nada poderia ser mais falso!

Não que essa extensão não acelere o processamento de vídeos... De fato, o faz. Mas, MMX, SSE (Streaming SIMD Extension) e AVX (Advanced Vector eXtension – a versão mais recente) nada têm haver com “multimedia”. E a coisa toda sequer é uma invenção da Intel: Nos anos 70 a *Texas Instruments* criou o conceito de SIMD (*Single Instruction, Multiple Data*). O nome diz tudo: Uma única instrução pode lidar com mais de um dado, ao contrário do que a maioria dos processadores fazem, normalmente.

Hoje em dia o MMX foi relegado a uma mera curiosidade. SSE e AVX, bem como outras extensões, são mais avançadas e possuem menos “problemas”. Para citar um, o MMX usa a pilha do ponto flutuante como se fossem registradores individuais, de forma que você **não** pode usar as instruções de ponto flutuante tradicionais e MMX sem que alguns cuidados sejam observados. Isso não acontece com SSE. Por causa disso, vou ignorar o MMX completamente e partir logo para o SSE. O caso do AVX é outro: Nem toda arquitetura x86 o suporta. Se bem que todos os processadores modernos (i3, i5 e i7) tendem a suportá-lo. Mostrarei as diferenças mais adiante.

Além das extensões MMV, SSE e AVX, outras foram incorporadas aos processadores: BMI (Bit Manipulation Instructions), FMA (Floating point Multyply and Add) e F16C (Floating point 16 bit Conversion).

## SSE

Para processar vários dados ao mesmo tempo, SIMD usa registradores especiais que contém conjuntos de dados. No caso da plataforma x86-64, temos 16 registradores que podem conter até 16 bytes (ou 8 'shorts', ou 4 'ints' ou 'floats', ou 2 'doubles' e até um grande número inteiro de 128 bits de tamanho). São os registradores XMM0 até XMM15:

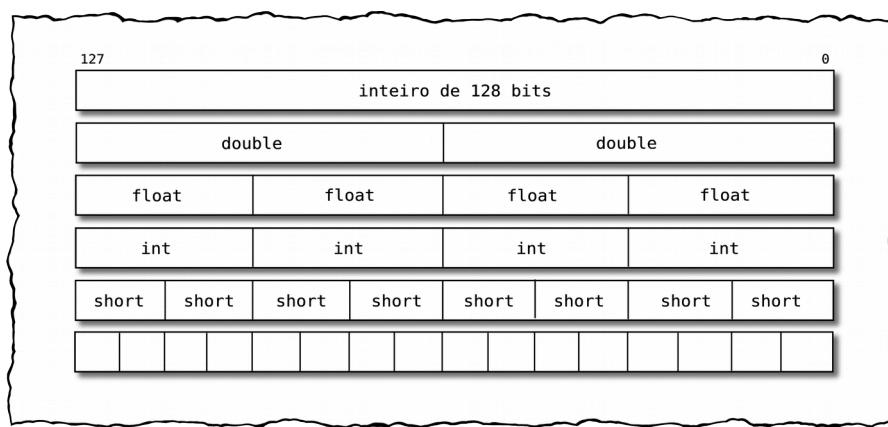


Figura 17: Capacidade dos registradores XMM (SSE).

SSE vem em diversos sabores. SSE, SSE2, SSSE3 (sim, são três 'S'), SSE4.1 e SSE4.2. Cada um adiciona ao anterior um conjunto de instruções especializadas.

## Funções “intrínsecas” para SSE.

A maneira de acessar esses registradores “especiais”, em C, é através de tipos especiais de classes de armazenamento. O tipo `_m128` (com dois ‘underscores’) equivale a um registrador SSE ou a 16 bytes, dependendo de onde o compilador escolher armazená-lo. Este tipo divide um registrador XMM em 4 floats. Temos ainda o tipo `_m128i` e `_m128d`. O primeiro é usado para lidar com os registradores XMM como se contivessem valores inteiros (inclusive de 128 bits) e o outro é usado para lidar dividir os registradores XMM em 2 *doubles*.

Outra vantagem de usar o tipo é que ele é automaticamente alinhado em 16 bytes. Esse alinhamento é importante, já que a carga de um registrador SSE através de um conjunto de dados desalinhados, na memória, consome, pelo menos, 1 ciclo de máquina adicional. O alinhamento também é importante com relação aos caches. Num cache com bloco de 64 bytes, podemos ter a carga de até 4 registradores SSE através de um único bloco de cache. Para exemplificar, a multiplicação de matrizes de 4 linhas e 4 colunas, onde cada elemento é um ‘float’, cabe completamente em dois blocos de cache (que estão na mesma linha!). Se você obter o endereço de uma variável do tipo `_m128` vai sempre ter os 4 bits menos significativos do endereço linear zerados:

```
...
__m128 x;
...
printf("0x%016lX\n", (unsigned long)&x);
...
```

O fragmento de código, acima, imprimirá algo como 0x0000000000601050. Não importa onde o tipo `_m128` esteja, esse ‘0’, nos bits menos significativos do endereço, sempre estarão zerados.

O problema com o tipo `_m128` é que operações simples não devem ser feitas diretamente. No GCC temos extensões que possibilitam usar operações básicas com o tipo `_v4si`, cujo apelido é `_m128`. Mas isso não é válido para outros compiladores. Você pode fazer algo assim, por exemplo:

```
__m128 x, y, z;
... /* carrega y e z de alguma forma... */
/* Funciona no GCC, mas é o jeito errado! */
x = y + z;
```

Pode até mesmo adicionar ou subtrair (ou multiplicar, ou dividir) constantes:

```
__m128 y, x = { 1, 2, 3, 4 };
...
y = x + 1; /* Faz: y = x + {1,1,1,1} */
```

Da maneira tradicional, para realizar operações, precisamos fazer uso de funções *intrínsecas*. Uma função intrínseca, no caso do SSE, é aquela que é traduzida diretamente para uma instrução em assembly. A operação de adição, acima, ficaria assim:

```
x = _mm_add_ps(y, z); /* Mesma coisa que x = y + z; */
```

Se você supor que o compilador escolha ‘x’ sendo `xmm0`, ‘y’ sendo `xmm1` e ‘z’ sendo `xmm2`, a linha acima será traduzida diretamente para:

```
movaps xmm0, xmm1
addps  xmm0, xmm2
```

O motivo para essa complicação está justamente na falta de informações que o compilador tem com relação a como os registradores XMM serão divididos, ao usar o tipo. Essa informação é codificada no próprio nome da função intrínseca (ou na instrução em assembly). No caso acima, ‘ps’ significa

“*packed singles*”, o que diz ao processador que dividiremos o registrador XMM em quatro *floats*. Se usássemos o sufixo ‘pd’ estaríamos usando o tipo dividido em dois “*doubles*” (de “*packed doubles*”). O sufixo começado com ‘s’ significa *scalar*, que quer dizer que apenas o primeiro componente do registrador será usado. ‘ss’ é “*scalar single*”, ou seja, um simples *float*.

Para usar funções intrínsecas de SSE no GCC basta fazer duas coisas: Incluir o header *x86intrin.h* nos seus códigos e dizer ao compilador qual é a versão do SSE que deseja usar através da opção “-msse”. SSE é usado por padrão na arquitetura x86-64, mas você pode escolher usar SSE2, SSE4.1 ou SSE4.2, se seu processador suportar. Para usar SSE4.2, por exemplo, basta usar a opção “-msse4.2”<sup>59</sup>.

## Exemplo do produto escalar

Eis um exemplo simples: Um produto escalar é a multiplicação, coordenada por coordenada, de um vetor. Numa aplicação matemática ou num *graphics engine* tridimensional poderíamos ter vetores de coordenadas (x, y, z). O produto vetorial então é definido como:

$$\vec{a} \cdot \vec{b} = (a_x \cdot b_x, a_y \cdot b_y, a_z \cdot b_z)$$

Isso pode ser traduzido numa função de maneira bem simples:

```
struct vector_s {
    float x, y, z;
};

float dot(struct vector_s *a, struct vector_s *b)
{
    return (a->x * b->x) + (a->y * b->y) + (a->z * b->z);
}
```

Compilando e verificando o código assembly gerado, com a máxima otimização, temos:

```
dot:
    movss xmm0, dword ptr [rdi]
    movss xmm1, dword ptr [rdi+4]
    mulss xmm0, dword ptr [rsi]
    mulss xmm1, dword ptr [rsi+4]
    addss xmm0, xmm1
    movss xmm1, dword ptr [rdi+8]
    mulss xmm1, dword ptr [rsi+8]
    addss xmm0, xmm1
    ret
```

A primeira implementação, usando SSE, que poderíamos fazer para acelerar um pouquinho as coisas é esta:

```
#include <x86intrin.h>

union xmm_u {
    struct {
        float x, y, z; /* Estou ignorando o 4º elemento aqui! */
    } s;
    __m128 x;
};

float dot_sse(__m128 a, __m128 b)
{
    union xmm_u u;

    u.x = _mm_mul_ps(a, b);
    return u.s.x + u.s.y + u.s.z;
}
```

---

<sup>59</sup> Uma excelente referência para as funções intrínsecas pode ser encontrada online em <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

O que melhora um pouco as coisas:

```
dot_sse:  
    mulps  xmm0,  xmm1  
    movaps xmmword ptr [rsp-24],  xmm0  
    movss  xmm0,  dword ptr [rsp-24]  
    addss  xmm0,  dword ptr [rsp-20]  
    addss  xmm0,  dword ptr [rsp-16]  
    ret
```

Existem alguns ajustes finos que podemos fazer para melhorar isso, mas o interessante mesmo é saber que existe uma instrução SSE que faz justamente um produto vetorial, se seu processador tiver suporte ao SSE 4.1:

```
float dot_sse41(__m128 a, __m128 b)  
{  
    return _mm_cvtss_f32(_mm_dp_ps(a, b, 0x71));  
}
```

A função intrinseca `_mm_dp_ps` multiplica cada parte de registradores SSE relacionados com a máscara dos 4 bits superiores do terceiro parâmetro e os soma. Depois, armazena o resultado nas posições dos 4 bits inferiores da máscara. Por isso o valor 0x71 (7 é a máscara para os 3 floats inferiores de 'a' e 'b'; e 1 é a máscara para colcar o resultado na posição 0 do registrador resultante).

A função intrinseca `_mm_cvtss_f32` pega o float na posição 0 do tipo `__m128` e retorna um 'float'. Isso é eliminado no código final, como pode ser visto abaixo, mas é necessário se quisermos lidar com o resultado como um tipo 'float' simples, em C:

```
dot_sse41:  
    dpps  xmm0,  xmm1,  113  
    ret
```

Nada mal, huh? De 8 instruções caímos para 5 e, finalmente, melhoramos para uma só. Isso poupa um bocado de espaço, mas essa otimização final é, essencialmente, o mesmo que a função anterior – talvez poupando um único ciclo de máquina... As duas últimas funções são bem melhores que a primeira. Enquanto a primeira toma entre 25 e 30 ciclos, as duas últimas tomam cerca de 14, de acordo com a documentação da Intel. Ou seja, as duas últimas funções tiveram um aumento de performance de cerca de 100% em relação à primeira.

### ***Uma “otimização” que falhou – o produto vetorial***

Num esquema normal de processamento, para calcular o produto vetorial de dois vetores tridimensionais (x,y,z) teríamos uma rotina deste tipo:

```
struct vector_s {  
    float x, y, z;  
};  
  
void cross(struct vector_s *vout, const struct vector_s *v1, const struct vector_s *v2)  
{  
    vout->x = (v1->y * v2->z) - (v1->z * v2->y);  
    vout->y = (v1->z * v2->x) - (v1->x * v2->z);  
    vout->z = (v1->x * v2->y) - (v1->y * v2->x);  
}
```

O código gerado, mesmo com otimização, é mais ou menos este:

```

cross:
    movss xmm3, dword ptr [rdx+8]
    movss xmm1, dword ptr [rsi+8]
    movss xmm0, dword ptr [rsi+4]
    movss xmm2, dword ptr [rdx+4]
    mulss xmm0, xmm3
    mulss xmm2, xmm1
    subss xmm0, xmm2
    movss dword ptr [rdi],xmm0
    movss xmm2, dword ptr [rdx]
    movss xmm0, dword ptr [rsi]
    mulss xmm1, xmm2
    mulss xmm3, xmm0
    subss xmm1, xmm3
    movss DWORD PTR [rdi+4],xmm1
    mulss xmm0, dword ptr [rdx+4]
    mulss xmm2, dword ptr [rsi+4]
    subss xmm0, xmm2
    movss dword ptr [rdi+8],xmm0
    ret

```

E aqui usaremos a mágica do SSE. Repare que cada componente é multiplado com outro diferente. Temos:

$$(y_1, z_1, x_1) \cdot (z_2, x_2, y_2) - (z_1, x_1, y_1) \cdot (y_2, z_2, x_2)$$

Então, tudo o que temos que fazer é “embaralhar” as coordenadas, multiplicá-las e depois subtraí-las. A rotina fica, assim:

```

#include <x86intrin.h>

__m128 cross_sse(__m128 v1, __m128 v2)
{
    return _mm_sub_ps(
        _mm_mul_ps(
            _mm_shuffle_ps(v1, v1, _MM_SHUFFLE(3,1,2,0)),
            _mm_shuffle_ps(v2, v2, _MM_SHUFFLE(3,2,0,1))
        ),
        _mm_mul_ps(
            _mm_shuffle_ps(v1, v1, _MM_SHUFFLE(3,2,0,1)),
            _mm_shuffle_ps(v2, v2, _MM_SHUFFLE(3,1,2,0))
        )
    );
}

```

Compilado, não? Mas ela parece ser mais rápida que a função *cross*:

```

cross_sse:
    movaps xmm3, xmm1
    movaps xmm2, xmm0
    shufps xmm3, xmm1, 216
    shufps xmm2, xmm0, 225
    shufps xmm1, xmm1, 225
    shufps xmm0, xmm0, 216
    mulps xmm2, xmm3
    mulps xmm0, xmm1
    subps xmm0, xmm2
    ret

```

Podemos, ainda, melhorar a rotina um pouco mais, eliminando um “shuffle”:

```

__m128 cross2_sse(__m128 v1, __m128 v2)
{
    __m128 r;

    r = _mm_sub_ps(
        _mm_mul_ps(v1,
                    _mm_shuffle_ps(v2, v2, _MM_SHUFFLE(3,0,2,1))),
        _mm_mul_ps(v2,
                    _mm_shuffle_ps(v1, v1, _MM_SHUFFLE(3,0,2,1)))
    );

    return _mm_shuffle_ps(r, r, _MM_SHUFFLE(3,0,2,1));
}

```

E o resultado parece ainda mais promissor:

```

cross2_sse:
    movaps xmm2,xmm0
    shufps xmm2,xmm0,201
    mulps  xmm2,xmm1
    shufps xmm1,xmm1,201
    mulps  xmm1,xmm0
    subps  xmm1,xmm2
    shufps xmm1,xmm1,201
    movaps xmm0,xmm1
    ret

```

Eis a pergunta de 1 milhão de dólares: Será que a função *cross2\_sse* é mais rápida que *cross\_sse*? E quanto a primeira rotina?

Por incrível que pareça as três rotinas são muito parecidas. A primeira gasta uns 127 ciclos e a última uns 121. Não é lá um ganho que valha à pena (uns 2.5%!), dada a complexidade aparente das funções que usam SSE! Eu manteria a última somente pelo fato dela ser menor.

### ***Uma otimização bem sucedida: Multiplicação de matrizes***

Dadas duas matrizes de 4 linhas e 4 colunas (column major, ao estilo OpenGL), a multiplicação delas produzirá uma terceira matriz. Se informarmos essas matrizes em forma de vetores de 16 floats, a rotina clássica para a multiplicação é a descrita na função *Matrix4x4Multiply*, e a versão SSE, otimizada está listada em *Matrix4x4MultiplySSE*:

```

/* Ao invés de usar arrays bidimensionais, uso arrays
   de 16 floats... */
void Matrix4x4Multiply(float *out, const float *a, const float *b)
{
    int row, col, k;

    for (row = 0; row < 4; row++)
        for (col = 0; col < 4; col++)
    {
        out[4*row+col] = 0.0f;

        for (k = 0; k < 4; k++)
            out[4*row+col] += a[4*k+col] + b[4*row+k]
    }
}

```

```

/* Usando um pouco de criatividade para lidar com os registradores
   de 128 bits do SSE, essa função faz a mesma coisa que a anterior! */
void Matrix4x4MultiplySSE(float *out, const float *a, const float *b)
{
    int i, j;
    __m128 a_column, b_column, r_column;

    for (i = 0; i < 16; i += 4)
    {
        b_column = _mm_set1_ps(b[i]);
        a_column = _mm_loadu_ps(a);
        r_column = _mm_mul_ps(a_column, b_column);

        for (j = 1, j < 4; j++)
        {
            b_column = _mm_set1_ps(b[i+j]);
            a_column = _mm_loadu_ps(&a[j*4]);
            r_column = _mm_add_ps(
                _mm_mul_ps(a_column, b_column),
                r_column
            );
        }
        _mm_storeu_ps(&out[i], r_column);
    }
}

```

A diferença de performance das duas funções chega a ser de quase 1000 ciclos de clock, onde a segunda é mais rápida. Em minhas medições a primeira gasta cerca de 2800 ciclos, enquanto a segunda, 1800. Ou seja, um aumento de performance de 55%!

## **E quando ao AVX?**

AVX (*Advanced Vector eXtension*) é, em essência, a mesma coisa que SSE, com registradores maiores... No AVX os registradores XMM são estendidos de 128 bits de tamanho para 256 bits e são renomeados para YMM.

Enquanto escrevo a Intel pretende lançar nova arquitetura de processadores que possuem suporte ao AVX-512... A mesma coisa que SSE e AVX, mas com registradores de 512 bits, chamados agora de ZMM.

Para AVX e AVX2 existem os tipos intrínsecos `_m256`, `_m256i` e `_m256d`. Eles funcionam da mesma maneira que os tipos `_m128` e seus derivados, no caso do SSE, só que suportam o dobro de componentes e, é claro, têm o dobro de tamanho. No AVX-512 temos o tipo `_m512` e os mesmos derivados.

## **Outras extensões úteis: BMI e FMA**

Tanto a Intel quanto a AMD estão, de tempos em tempos, adicionando novas instruções em seus processadores e fazendo disso um padrão da indústria. A extensão BMI lida com manipulação de bits (*Bit Manipulation Instructions*) e a FMA faz um mistureba de operações em ponto flutuante (*Fused Multiply and Add Instructions*).

A primeira extensão, BMI, foi criada para aglutinar, em uma única instrução, coisas que podem ser feitas com conjuntos de instruções como AND, OR e SHL (ou SHR). Por exemplo, se quisermos zerar todos os bits a partir da posição 31 de RDI, sem alterar seu valor e colocando o resultado em RAX podemos fazer:

```

mov  rax,rdi
and  rax,0x3fffffff      ; usando AND com uma máscara

```

A máscara é confusa, já que temos que contar as posições dos bits a partir de zero. A instrução

BZHI torna isso mais fácil, mas não mais performático:

```
mov rdx,30
bzhi rax,rdx,rdx ; RDI terá seus bits a partir de RDX zerados e tudo copiado para RAX.
```

Fizemos, essencialmente, a mesma coisa que o AND faz, mas BZHI é mais lenta, já que ela usa o um prefixo.

Isso não quer dizer que BMI seja inútil. Considere essa outra necessidade: Suponha que você queira contar quantos bits estejam setados num registrador. Com instruções tradicionais poderíamos fazer algo assim:

```
; Protótipo: unsigned int popcnt(unsigned long x);
popcnt:
    xor    eax,eax
    mov    edx,64
.loop:
    lea    ecx,[rax+1]
    test   dli,1
    cmovne eax,ecx
    shr    rdi,1
    dec    rdx
    jnz    .loop
    ret
```

A extensão BMI vem a calhar, neste caso, porque podemos fazer isso com apenas uma instrução e sem alterarmos o conteúdo de RCX,RDX e RDI:

```
popcnt rax,rdi
ret
```

BMI também possui instruções interessantes para extração de bits e manipulações lógicas “fundidas” (como ANDN, onde é feito um AND seguido de um NOT)... Para habilitar o uso de BMI no GCC, se seu processador suportar, use as opções -mbmi ou -mbmi2.

A extensão FMA adiciona extensões ao SSE e ao AVX para lidar com equações lineares, do tipo:

$$y = a \cdot x + b$$

Ao invés de usarmos duas instruções, uma para multiplicar e outra para somar, as duas estão “fundidas” (fused) numa mesma instrução. Se você habilitar FMA no gcc, via opções -mfma ou -mfma4, sempre que compilador topar com uma equação linear em ponto flutuante, ele tentará usar a extensão, gerando códigos menores e, possivelmente, mais rápidos. Se tivermos uma função do tipo:

```
float ma(float a, float x, float b) { return a*x + b; }
```

A diferença das duas versões (com e sem FMA) seria assim:

```
; versão 1
ma:
    mulss xmm0,xmm1
    addss xmm0,xmm2
    ret
-----%<---- corte aqui -----%>-----
; versão 2 (fma)
ma:
    vfmaddss xmm0, xmm0, xmm1, xmm2
    ret
```

Suspeito que a diferença real seja apenas uma pressão menor nos caches, já que a segunda versão gera código menor:

```
$ objdump -d -M intel-mnemonic test1.o
test1.o:      file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <ma>:
 0:  f3 0f 59 c1          mulss  xmm0, xmm1
 4:  f3 0f 58 c2          addss  xmm0, xmm2
 8:  c3                   ret

$ objdump -d -M intel-mnemonic test2.o
test2.o:      file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <ma>:
 0:  c4 e3 f9 6a c2 10    vfmaddss xmm0, xmm0, xmm1, xmm2
 6:  c3                   ret
```

Claro que para funções mais complicadas, tendo uma função especializada para calcular equações lineares, evita muitas movimentações de registradores XMM para temporários. Potencialmente aumentando a performance do código final.



# Capítulo 14: Dicas e macetes

Este capítulo é apenas uma coletânea de dicas e macetes que ofereço a você, a respeito de aspectos que podem ser importantes para que certos erros não sejam cometidos, ou até mesmo, coisas interessantes, das quais você pode não ter pensado...

## Valores booleanos: Algumas dicas interessantes em C

Numa comparação, na linguagem C, um valor zero é sempre considerado falso. Já um valor diferente de zero é sempre verdadeiro. E, ao mesmo tempo, o tipo *int* é o tipo padrão para a avaliação de expressões booleanas, no entanto, a regra do zero e do não-zero vale para qualquer tipo, incluindo ponteiros. O símbolo *NULL*, por exemplo, é definido como sendo um ponteiro *void* que aponta para 0.

Acontece que, ao avaliar uma expressão booleana, o compilador atribui apenas um de dois valores: 0 para falso e 1 para verdadeiro. **Isso é parte da especificação da linguagem e é um recurso garantido.** Pode ser demonstrado com a chamada abaixo:

```
printf("true = %d\nfalse = %d", (10 == 10), (0 == 1));
```

Isso imprimirá os valores 1 e 0, nesta ordem.

O compilador pode ignorar a avaliação e se ater apenas à primeira regra e comparar apenas contra o valor zero, se isso criar código mais performático, mas se você quiser usar o resultado de uma expressão booleana os valores 1 e 0 são, como já disse, garantidos. Por exemplo, as instruções abaixo são equivalentes:

```
if (x < 2) y++; /* adiciona 1 a y se x < 2. */
y += (x < 2); /* adiciona 1 a y se x < 2! */
```

Outra dica legal é o uso do operador booleano *!* duas vezes. O operador *!* serve para inverter o valor booleano que o segue, se a expressão à direita da exclamação for 0 o resultado será 1 e vice-versa. Ao usar *!!* manteremos o valor booleano original (mas, se tivermos originalmente um valor diferente de zero, mas que não seja 1, então o resultado óbvio será 1)... Assim, para evitar que o compilador emita avisos em construções como:

```
/* Equivale a 'if ((f = fopen(...)) != NULL) ...' */
if (f = fopen("file.dat", "r")) ...
```

Neste caso o compilador avisa que uma atribuição (válida!) foi colocada no critério do *if...* Esse é um “erro” comum, então o *gcc* avisa... mas, se for isso mesmo que você quer fazer, basta colocar dois *!!*, assim:

```
if (!! (f = fopen("file.dat", "r"))) ...
```

Se *f* for *NULL* o operador *!!* avaliará a expressão para 0 (falso), senão para 1 (verdadeiro). E nenhum aviso é dado. A mesma coisa pode ser feita com o exemplo anterior (aquele, ai em cima, onde incrementamos *y*), mas se quisermos comparar *x* com *true*, por exemplo:

```
if (x) y++; /* Incrementa y se x for 'verdadeiro'. */
y += !!x; /* Faz a mesma coisa! */
```

## **Quanto mais as coisas mudam...**

Não fique surpreso em constatar que as duas instruções abaixo são executadas em um número diferente de ciclos de máquina:

```
add rax,[var]
add [var],rax ; Essa é duas vezes mais lenta que a anterior.
```

A segunda instrução efetua uma leitura do endereço da variável 'var', faz a adição e grava o resultado de volta na posição de 'var'. A primeira faz apenas uma leitura. O comportamento do primeiro caso é conhecido como *leitura-modificação-escrita*. Isso sempre adiciona ciclos de máquina ao processamento da instrução e é um comportamento que deve ser evitado.

Outra coisa interessante é que, das sequências:

```
cmp qword [var],0
-----%<---- corte aqui -----%>-----
xor rax,rax
cmp [var],rax
```

A sequência XOR/CMP é mais rápida. O motivo é que a instrução de comparação com um valor imediato do tamanho de um QWORD é muito grande. Quanto maior a instrução, mais lenta ela é. O par de instruções XOR/CMP, onde CMP não está usando um valor imediato, ocupa menos espaço.

## **INC e DEC são lerdas!**

O que supreende é que algumas instruções têm comportamento de *leitura-modificação-escrita* mesmo que seu operando não seja uma referência à memória. É o caso de INC e DEC. Essas instruções fazem alterações parciais no registrador EFLAGS (elas não afetam o flag de Carry) e isso causa uma leitura de EFLAGS, modificação e escrita, adicionando ciclos. É por esse motivo que um bom compilador C prefere usar as instruções ADD e SUB, que escrevem sobre os flags diretamente<sup>60</sup>.

Existe ou outro problema com DEC (e, provavelmente, INC). No modo x86-64 não é possível usá-la com registradores de 16 ou 32 bits. Instruções como:

```
dec ecx
```

Não existem nesse modo! Isso acontece porque o micro-código usado para a instrução é usado para outra coisa! Você pode usar DEC com registradores como CL e RCX, mas não as outras variantes.

## **Não faça isso!!!**

A Intel recomenda que toda instrução RET seja “emparelhada” com uma função CALL correspondente. O que ela quer dizer com isso é que, para uma chamada feita com CALL, deverá existir uma instrução RET que retorne da chamada. Isso acontece naturalmente quando criamos nossas funções em C, mas em certos casos a coisa pode não parecer tão óbvia:

```
int zeromem(void *dest, size_t size) { return memset(dest, 0, size); }
```

Seu compilador provavelmente criará o seguinte código:

```
zeromem:
    mov rdx,rsi
    xor esi,esi
    jmp memset
```

---

<sup>60</sup> Estou falando apenas dos flags ligados a operações aritméticas. Flags D e I, por exemplo, são mantidos num outro circuito, mesmo que sejam mapeados em EFLAGS.

A instrução RET, que está emparelhada com o CALL que chamou essa rotina, está na função *memset*. Isso está ok... Mas, se você fizer uma coisa esquisita dessas, para chamar uma função:

```
mov rax, zeromem
push rax
ret           ; "Retornará" para a função zeromem!
```

Você já não terá um RET emparelhado com o um CALL, mas sim dois RETs (o que está listado acima e o que está na função *zeromem*). Essa técnica funciona, mas causará vários ciclos de máquina de penalidade em grande parte do seu código, já que deixará o *branch prediction* confuso. Além do que, a instrução RET é mais lenta que a instrução CALL...

Outra técnica comum, quando se quer obter o valor de RIP é esta:

```
call here
here:
pop rax      ; Pega o RIP armazenado na pilha!
...
```

Como sumimos com o RET que era esperado, já que o eliminamos para obter o conteúdo de RIP que estava na pilha, o *branch prediction* sofrerá com isso. Se você realmente precisar fazer algo desse tipo, eis um código melhor:

```
call here
; Neste ponto RAX conterá o valor de RIP.
...
here:
mov rax, [rsp]
ret          ; RET, emparelhado com o CALL.
```

## Aritmética inteira de multipla precisão

Uma vez um grande amigo, entusiasmado com a linguagem LISP, me provocou com o seguinte código, me desafiando a fazer o mesmo em C:

```
; ; Imprime o valor de 42424242.
(expt 4242 4242)
```

Esse código “simples” em C é impossível, já que os tipos inteiros têm, no máximo, 64 bits de tamanho. O resultado da operação acima te mostrará um grande número inteiro de 15390 algarismos e, com 64 bits, o valor máximo que podemos obter, sem overflow, é algo como  $1,8 \cdot 10^{19}$ . Ou seja, cerca de 20 algarismos.

Acontece que, em assembly, o flag CF e algumas instruções aritméticas, nos permitem lidar com **aritmética de multipla precisão**. Por exemplo: Se tivéssemos dois valores de 128 bits e quiséssemos somá-los, poderíamos fazer algo assim:

```
mov rax, [valor1]
mov rdx, [valor1+8]    ; coloca em RDX:RAX os 128 bits de "valor1".
add rax, [valor2]
adc rdx, [valor2+8]    ; Note que RDX é adicionado com a parte superior de "valor2" e
                       ; com o carry, vindo da adição anterior.
```

Usando o flag CF para obter os overflows das adições parciais, inferiores, podemos realizar adições com a quantidade de bits que quisermos. O mesmo acontece com subtrações: Existe uma instrução SBB (*Subtract with Borrow*), onde o flag CF é usado como “emprestimo”. E, com um pouquinho de trabalho podemos extrapolar algo semelhante para multiplicações e divisões.

No entanto, esse artifício só pode ser alcançado em assembly. Em C não há como, diretamente, acessar os flags. A linguagem esconde isso. Para tanto, existem bibliotecas especializadas para lidar

com múltipla precisão... Uma delas é a *libgmp* (GMP é acrônimo de *GNU Multiple Precision*). A rotina em LISP poderia ser escrita, em C, assim:

```
/* Calcula exp(4242,4242), como em lisp. */
#include <stdio.h>
#include <gmp.h>

int main(void)
{
    mpz_t r, s;

    mpz_init(r);
    mpz_init(s);

    mpz_set_ui(s, 4242); /* s = 4242; */
    mpz_pow_ui(r, s, 4242); /* r = pow(s, 4242); */

    mpz_clear(s); /* não precisamos mais do 's' */

    /* printf() não suporta o tipo mpz_t.
       Por isso usamos um printf especializado. */
    gmp_printf("%zd\n", r);

    mpz_clear(r); /* não precisamos mais do 'r' */

    return 0;
}
```

Mas, atenção! A biblioteca *libgmp* provavelmente não usa aritmética inteira em seu interior. LISP, é provável, também não! Esse tipo de biblioteca costuma usar *strings* como containers para os valores que serão manipulados. Dessa forma, as variáveis do tipo *mpz\_t* são limitadas apenas pela quantidade de memória disponível para a aplicação.

LibGMP é capaz de calcular o valor de  $\pi$ , pelo método de Chudnovsky, com 1 milhão de “casas decimais” em apenas 0,38 segundos, 10 milhões em 7 segundos, 100 milhões em 100 segundos e 1 bilhão em meia-hora. Mas, não se entusiasme tanto assim com essa biblioteca... Ela consome muita memória e é extremamente mais lenta que a aritmética inteira (e também em ponto flutuante) nativa do processador. Além de ser mais “complicada” de usar... A não ser que você tenha um bom motivo – como, por exemplo, querer calcular o ângulo de reentrada de uma sonda que você mandou para Marte – é recomendável que **não use** aritmética de multipla precisão.

## Você já não está cansado disso?

Provavelmente você já viu um código parecido com o abaixo:

```
FILE *f;
void *ptr;

if ((f = fopen(filename, "rt")) != NULL)
{
    if ((ptr = malloc(size)) == NULL)
    {
        fclose(f);
        return 1;
    }

    ...

    if (erro)
    {
        free(ptr);
        fclose(f);
        return 2;
    }
}
```

```

...
    free(ptr);
    fclose(f);
}

return 0;

```

Não acha que existem muitas chamadas para *fclose*? É claro que podemos reduzir isso com um *goto* bem posicionado:

```

FILE *f;
void *ptr;
int retcode = 0;

if ((f = fopen(filename, "rt")) != NULL)
{
    if ((ptr = malloc(size)) == NULL)
    {
        recode = 1;
        goto endOfFunction;
    }

    ...

    if (erro)
    {
        free(ptr);
        retcode = 2;
        goto endOfFunction;
    }

    ...

    free(ptr);
endOfFunction:
    fclose(f);
}

return retcode;

```

Mas se você é um daqueles chatos que consideram *goto* a pior coisa desde a invenção da axé music, então eis uma outra maneira. Podemos colocar um tratador de erros no começo de nossas funções e simplesmente saltar para ela, mas sem usar *goto*:

```

FILE *f;
void *ptr;
jmp_buf jb;
int retcode;

/* Ponto de salto, em caso de erro. */
if (retcode = setjmp(jb))
{
    if (retcode == 2)
        free(ptr);
    fclose(f);
    return retcode;
}

if ((f = fopen(filename, "rt")) != NULL)
{
    if ((ptr = malloc(size)) == NULL)
        longjmp(jb, 1);

    ...

    if (erro)
        longjmp(jb, 2);

    ...
}

```

```

        free(ptr);
        fclose(f);
    }

    return 0;
}

```

Ok, *longjmp* é um goto disfarçado, mas o código fica mais elegante, não? É como se registrássemos uma rotina de erros e depois coloquemos o código para funcionar (e não foi isso que fiz?!).

## Otimização de preenchimento de arrays

Às vezes algumas rotinas clássicas apresentam problemas interessantes. Um desses problemas é: “Como preencher um array da maneira mais rápida possível?”. Para responder isso vamos construir uma rotina que preenche um array com zeros. No nosso exemplo esse array tem tamanho variável e é do tipo *char*:

```
char array[n];
```

Só que queremos fazer nossa rotina o mais flexível possível, portanto o tipo do array deve ser irrelevante. Isso significa que usaremos ponteiros *void*. Ainda, 'n' pode ser qualquer tamanho. Pode ser 1, mas pode ser 32748235 também...

A rotina óbvia é essa:

```

void zerofill(void *ptr, size_t size)
{
    char *pc;
    int i;

    pc = ptr;
    for (i = 0; i < size; i++)
        *pc++ = 0;
}

-----%<---- corte aqui -----%<-----
; Código gerado pelo compilador.

zerofill:
    lea    rax,[rdi+rsi]           ; RAX contém agora o ponteiro além do final do array.
    test   rsi,rsi                ; Se 'size' == 0, sai da função.
    je     .L1
.L2:
    add    rdi,1                 ; ptr++;
    mov    byte ptr [rdi-1],0      ; *(ptr - 1) = 0;
    cmp    rdi,rax                ; Chegamos ao final do array?
    jne    .L2                   ; ... se não chegamos, volta ao loop.
.L1:
    ret

```

Esse código ai em cima é gerado com a opção *-O2* do compilador. Ao compilarmos com máxima otimização temos uma surpresa:

```

zerofill:
    test rsi,rsi
    je   .L1
    mov  rdx,rsi
    xor  esi,esi
    jmp  memset
.L1:
    ret

```

Descobrimos que *memset* é uma daquelas funções intrinsecas do compilador. Ao perceber o preenchimento de um array ele simplesmente substituiu todo o nosso código por uma chamada do tipo:

```
if (size != 0)
    memset(ptr, 0, size);
```

Vamos ignorar essa violação de nossa vontade e nos concentrar na rotina original... O problema com ela é que preenche um único byte de cada vez. Uma instrução MOV é executada em um ciclo de máquina **independente** do tamanho do operando. Assim, poderíamos preencher o array com uma QWORD por vez e o restante do array (que pode ter nenhum até 7 bytes adicionais) com tipos menores. A rotina fica mais complicada e maior, mas em muitos casos é mais performática:

```
void zerofill2(void *ptr, size_t size)
{
    size_t qsize;
    unsigned long *qptr;
    int I;

    qsize = size / sizeof(unsigned long);
    size = size % sizeof(unsigned long);
    qptr = ptr;

    while (qsize--)
        *qptr++ = 0;
    ptr = qptr;

    if (size >= sizeof(unsigned int))
    {
        *(unsigned int *)ptr = 0;
        ptr += sizeof(unsigned int);
        size -= sizeof(unsigned int);
    }

    if (size >= sizeof(unsigned short))
    {
        *(unsigned short *)ptr = 0;
        ptr += sizeof(unsigned short);
        size -= sizeof(unsigned short);
    }

    if (size > 0)
        *(unsigned char *)ptr = 0;
}
```

O código acima é muito bom para arrays com tamanhos maiores que 8. Mas, eis um código melhor:

```
; zfill.asm
bits 64
section .text

;
; RDI = ptr, RSI = size
;
global zerofill3:function
align 16
zerofill3:
    xor eax,eax
    mov rcx,rsi
    rep stosb      ; preenche byte por byte?!?!
    ret
```

E esse é um exemplo de uma coisa que era boa, passou a ser ruim por muito tempo e voltou a ser boa, de novo! A instrução STOSB foi feita para armazenar o valor do registrador AL no endereço apontado por RDI. Ao acrescentar o prefixo REP a instrução armazena AL no bloco de memória cujo endereço base é dado por RDI e tem RCX bytes de tamanho. Com uma pegadinha: Nas arquiteturas modernas (Nehalem e superiores) o processador procurará usar o “macete” de armazenar o maior tamanho possível primeiro.

A mesma coisa funciona para REP MOVSB. As funções REP CMPSB e REP SCASB, parecem, continuam sendo exceções a essa regra... elas ainda são ruins, em termos de performance. Mas, não confie em mim, é bom medir:

Função	Ciclos	Ganho
zerofill	231523	-
zerofill2	20910	1007%
zerofill3	5219	4336%

Tabela 10: Performances das versões de zerofill.

Só modificando o jeito de preencher um array (de bloco em bloco), melhoramos a performance em mais de 11 vezes. Usando um recurso do processador, melhoramos em 44!

**Aviso:** Uma maneira de verificar se seu processador suporta LODSB/STOSB/MOVSB rápidos é com essa pequena rotina:

```
; Protótipo:
;   int test_fast_block_operations(void);
; Função equivalente, em C:
;   int test_fast_block_operations(void)
;
{
;   int a,b,c,d;
;   _cpuid(7, a, b, c, d);
;   return (b & 0x200) != 0;
}
test_fast_block_operations:
push rbx
mov eax,7
cpuid
xor eax,eax
test ebx,0b10_0000_0000    ; bit 9 de EBX indica essa feature.
setz al
pop rbx
ret
```

### Tentando otimizar o RFC 1071 check sum. E falhando...

Em um projeto envolvendo sockets, precisei usar a rotina de cálculo de *check sum* estabelecida pela RFC 1071. “O que é menos complicado do que calcular *check sums*?", você deve estar pensando. Bem, eis a função, como implementada pela RFC:

```
unsigned short cksum(void *addr, size_t count)
{
    unsigned int sum;
    unsigned short *p;

    /* Condição em que a rotina funciona bem. */
    assert(count <= 131072);

    sum = 0;
    p = addr;
    while ( count > 1 )
    {
        sum += *p++;
        count -= sizeof(unsigned short);
    }

    if ( count > 0 )
        sum += *(unsigned char *)p;

    while (sum>>16)
        sum = (sum & 0xffff) + (sum >> 16);

    return ~sum;
}
```

A rotina usa uma variável de 32 bits para, no primeiro loop, acumular porções de 16 bits de tamanho, obtidas do buffer. Se sobrar um byte que ainda não foi acumulado, então a rotina o faz. E, como passo final, os transportes (a parte dos 16 bits superiores do acumulador) são acrescentados a

ele para que obtenhamos apenas os 16 bits inferiores.

Tudo funciona muito bem se obtivermos até 65535 “transportes” (todos os 16 bits superiores do acumulador setados). Qualquer número de transportes adicionais e a rotina devolverá o *check sum* errado. Dessa forma, a rotina aceita lidar com buffers de até 128 KiB de tamanho (65535 transportes vezes 2 bytes). A linha com a assertiva está lá para, durante o debugging, verificarmos se essa condição foi violada.

Uma maneira de acelerar as coisas e resolver a limitação do tamanho do buffer, em teoria, é usar os recursos que temos em mãos. No caso, registradores de 64 bits. A rotina abaixo só tem um único loop que lê *unsigned long's* de cada vez, levando em conta os transportes parciais. Ela usa um artifício perigoso... Depois do loop, se houverem bytes adicionais, ela lê um último *unsigned long* do buffer, mas zera os bits que não estão presentes. É perigoso porque estamos lendo além do tamanho total do buffer, mas ignorando o que não deveríamos ter lido.

No final da rotina acumulamos os 32 e 16 bits parciais do acumulador, levando o transporte em consideração:

```
unsigned short cksum2(void *data, size_t count)
{
    unsigned long sum, tmp, *p;
    unsigned int u1, u2;
    unsigned short s1, s2;
    size_t ulcount;
    int shift;

    sum = 0;
    p = data;
    ulcount = count >> 3;
    count -= ulcount << 3;

    while (ulcount--)
    {
        sum += *p;
        if (sum < *p) sum++;
        p++;
    }

    if (count)
    {
        shift = (8 - count) * 8;
        tmp = *p & (0xffffffffffffffffUL >> shift); /* perigoso, mas funciona! */
        sum += tmp;
        if (sum < tmp) sum++;
    }

    u1 = sum;
    u2 = sum >> 32;
    u1 += u2;
    if (u1 < u2) u1++;

    s1 = u1;
    s2 = u1 >> 16;
    s1 += s2;
    if (s1 < s2) s1++;

    return ~s1;
}
```

Era de se esperar que *cksum2* fosse, pelo menos, duas vezes mais rápida que a rotina original, já que estamos lendo duas “mais rápido”, no loop principal. Mas não é isso o que acontece. Eis a medição dos ciclos gastos, usando um buffer de 32775 bytes<sup>61</sup>:

---

61 Como condição de testes o buffer deve ter 7 bytes a mais do que o múltiplo de 8. Assim, teremos que ler (n+1) “unsigned long’s. Onde o último contém os 7 bytes restantes...

Função	Ciclos
cksum	6696
cksum2	8648 (perda de 29%!)

Tabela 11: Consumo de ciclos de cksum e cksum2.

Essa é uma daquelas funções onde criar um código assembly não adianta muita coisa... Tentei criar uma rotina usando alguns macetes (como acumular 32 bytes de cada vez, na iteração do loop inicial) e o que consegui foi uma performance ligeiramente superior à *cksum2* (somente cerca de 1% de ganho!).

Na arquitetura x86-64, o bom e velho *cksum* da RFC 1071 ainda é muito bom!

### Previsão de saltos e um array de valores aleatórios

Esse problema interessante me foi indicado por um amigo: A rotina abaixo preenche um array de 32768 *ints* com valores aleatórios entre 0 e 255. Depois todos os valores do array são somados. A soma dos itens do array é feita 100 mil vezes para facilitar a medição do tempo. O código original usa a função *clock* para medir a performance, em segundos:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_ITERATIONS 100000
#define MAX_BUFFER_SIZE 32768

int buffer[MAX_BUFFER_SIZE];

/* Preenche array com valores aleatórios entre 0 e 255. */
void random_fill_buffer(void)
{
    int i;

    /* Não alimento o 'seed' aqui para obter a mesma sequência "aleatória" toda vez! */
    for (i = 0; i < MAX_BUFFER_SIZE; i++)
        buffer[i] = rand() % 256;
}

/* Usada por qsort. */
int compare(const void *x, const void *y) { return *(int *)x < *(int *)y; }

int main(void)
{
    int i, j;
    long sum;
    clock_t clkstart;
    double clkelapsed;

    random_fill_buffer();

    // Retire isso daqui para testar com o array fora de ordem.
    qsort(buffer, MAX_BUFFER_SIZE, sizeof(int), compare);

    clkstart = clock();
    for (i = 0; i < MAX_ITERATIONS; i++)
        for (j = 0; j < MAX_BUFFER_SIZE; j++)
            if (buffer[j] >= 128)
                sum += buffer[j];

    clkelapsed = (double)(clock() - clkstart) / CLOCKS_PER_SEC;

    printf("sum = %ld.\n"
           "Tempo decorrido: %.3f segundos.\n", sum, clkelapsed);

    return 0;
}
```

Eis a tabela com o tempo de execução medido com a ordenação do array e sem ordenação do array, bem como com níveis diferentes de otimização:

	<b>-O0</b>	<b>-O1</b>	<b>-O2</b>	<b>-O3</b>
<b>Array ordenado</b>	6.49 segundos	1.76 segundos	1.74 segundos	2.62 segundos
<b>Array não ordenado</b>	17.34 segundos	10.61 segundos	10.26 segundos	2.62 segundos

Tabela 12: Tempos de execução na varredura em `SortedArrayTest`.

Os níveis de otimização 1 e 2 obtiveram um tempo muito bom com o array ordenado, mas a rotina é uma porcaria quando os itens são aleatórios. O nível 3 manteve a varredura do array em tempo constante e, é claro, o sem otimizações tivemos os piores resultados.

O que acontece nos níveis 2 e 3, por exemplo?

O problema da rotina é o 'if' dentro do loop que acumula o conteúdo dos itens do array na variável 'sum':

```
if (data[i] >= 128)
    sum += data[i];
```

Um 'if' sempre executa um salto condicional. No exemplo, o conjunto de instruções que obtém o valor de 'data[i]' e somam com o conteúdo de 'sum', colocando de volta em 'sum', é omitido se 'data[i]' for menor que 128. A versão do código deste 'if', gerado pela otimização de nível 2, é mais ou menos assim:

```
; Considere: RDX = &data[i]
;           EBX = sum
...
movsx rcx,dword [rdx]
cmp   rcx,127
jle   1f
add   ebx,ecx
1:
...
```

Sempre que 'data[i]' for menor ou igual a 127 um salto é feito, evitando a acumulação.

Acontece que desde os 486s os processadores da arquitetura 80x86 decidificam as instruções bem antes delas serem executadas. No caso de saltos condicionais o processador tem que "adivinhar" se o salto será tomado ou não **antes** de saber o estado dos flags. Isso é feito por uma "rotina" interna do processador chamada *branch prediction*. O processador mantém uma estatística sobre os saltos adivinhados que aconteceram ou não, na tentativa de melhorar a previsão.

Na primeira vez que o processo passa pelo 'if' e se 'data[i]' for, de fato, maior ou igual a 128, o salto **não** será feito. O processador assume que da próxima vez que encontrar esse salto condicional ele também não saltará. Mas, se isso não acontecer, houve um erro na adivinhação. O processador terá que voltar atrás e decodificar as instruções que deixou de fazê-lo, já que o salto aconteceu de fato. Além de atualizar as estatísticas do *branch prediction*! Isso toma ciclos adicionais...

Numa sequência ordenada, 99% dos saltos serão previstos corretamente. Numa sequência fora de ordem a previsão pode estar errada em 99% dos casos!

Mas, o que o nível 3 de otimização faz que consegue obter tempo constante em ambos os casos? Ele se livra do salto condicional!

```

; Considere: RDX = &data[i]
;           EBX = sum
;           R8  = tmp_sum;
...
    mov     ecx, [rdx]
    movsx  r8,ecx
    add    r8,rbx
    cmp    ecx,128
    cmovge rbx,r8
...

```

O compilador fez um trabalho interessante aqui. R8 só será movido para RBX se, e somente se, ECX for maior ou igual a 128. A instrução CMOVcc gasta sempre 1 ciclo de máquina, mesmo quando a condição não é satisfeita. E note que o salto condicional sumiu!

A dica aqui é: Elimine os saltos condicionais tanto quanto possível. E, ainda, verifique se seu compilador fez isso!

### **Usar raiz quadrada, via SSE, parece estranho...**

Especialmente no modo x86-64, o compilador C usará SSE para lidar com aritmética em ponto flutuante. Isso já foi discutido anteriormente. Você topará com um código bem estranho no caso do uso de raiz quadrada:

```

/* Simples exemplo de código em C */
#include <math.h>

float get_sqrt(float x)
{
    return sqrtf(x);
}
----- cortar aqui -----
; Código equivalente, em assembly:
get_sqrt:
    sqrtss  xmm1,xmm0
    ucomiss xmm1,xmm1
    jp       .L1
    movaps  xmm0,xmm1
    ret
.L1:
    push    rax
    call    sqrtf
    pop    rdx
    ret

```

A primeira “estranheza” é por que o compilador incluiu essa comparação e uma chamada à função *sqrtf* da libc? O problema está em alguns valores usados nos operadores da instrução SQRTSS.

As regras para lidarmos com resultados NaNs e infinitos no processador são diferentes das regras tradicionais da função *sqrtf* e, por isso, o compilador coloca esse código adicional<sup>62</sup>. Ele testa pelo resultado de SQRTSS e, se o flag de paridade PF estiver setado, significa que houve um problema. Daí o código irá usar a função *sqrtf* para obter a raiz quadrada.

O que me causa outra estranheza é a necessidade do compilador salvar *rax* e recuperá-lo em *rdx*, já que ambos os registradores não precisam ser preservados na convenção de chamada x86-64. Em revisões futuras deste livro, se eu achar uma explicação para esse fato, comentarei sobre o assunto. Por enquanto, essa necessidade, para mim, ainda é um mistério.

Poderíamos usar as funções intrínsecas do compilador para SSE e obter um código menor:

---

<sup>62</sup> O processador usa “tiops” diferentes de NaNs: SNaNs e QNaNs. Onde o primeiro é mais “drástico” que o segundo.

```

#include <x86intrin.h>

float get_sqrt(float x)
{
    return _mm_cvtss_f32(_mm_sqrt_ss(_mm_set_ss(x)));
}
----%----- corte aqui ----%-----
; Código equivalente em assembly
get_sqrt:
    movss [rsp-12],xmm0      ; WTF?!
    movss xmm0,[rsp-12]       ;
    sqrtss xmm0,xmm0
    ret

```

Embora a rotina final fique menor e potencialmente mais rápida (mas nem tanto!), essa necessidade de garantir que os *floats* superiores estejam zerados me parece preciosismo do compilador. E, em C, não há muito o que fazer para evitar isso.

## **Gerando números aleatórios**

Às vezes precisamos obter valores aleatórios em nossas aplicações. A maneira mais simples de fazê-lo é usando uma função da *libc*: *rand()*. Ela é declarada em *stdlib.h* como:

```
int rand(void);
```

E a documentação nos diz que a função retorna um valor entre 0 e RAND\_MAX, que é tipicamente definida como INT\_MAX, ou seja, 0x7fffffff. Mas essa função tem dois problemas.

O primeiro problema é a maneira como o valor aleatório é gerado... A maioria das funções de geração de valores aleatórios que existem por ai usam o que se chama de *Gerador de Congruência Linear*. É um nome complicado para uma “fórmula”:

$$X_{n+1} = (aX + b) \bmod m$$

Onde os valores de 'a', 'b' e 'm' são fixos, pré-calculados. O valor aleatório  $X_{n+1}$  depende exclusivamente do valor inicial  $X_n$ . Esse valor inicial,  $X_0$ , é conhecido como “semente aleatória” (*random seed*). Eis a implementação, simplificada, de *rand()*, no código da glibc que tenho em mãos:

```
return (seed = (seed * 1103515245 + 12345) % 0x7fffffff);
```

O valor de 'seed' inicial é também escolhido à dedo para parecer que *rand()* seja aleatório, se uma “semente” não for fornecida, mas é sempre o mesmo. A função *srand()*, que fornece a semente para o gerador de congruência linear somente atualiza essa variável estática.

Assim, a função *rand()* é “pseudo-aleatória”. Felizmente a arquitetura *Haswell* implementa um gerador aleatório “de verdade”, usando um fenômeno quântico como fonte de entropia... Isso tudo dentro do seu processador! Através de uma única instrução podemos obter o número aleatório que quisermos que é **garantido** ser aleatório. Trata-se da função RDRAND:

```

; random.asm
bits 64
section .text

; Protótipo:
;     unsigned long getrandom(void);
global getrandom:function
getrandom:
    rdrand rax
    jnc    getrandom
    ret

```

A instrução RDRAND, quando falha, retorna com o flag Carry zerado. Tudo o que temos que fazer

é executar RDRAND novamente. A desvantagem é que RDRAND pode ser lenta... A documentação da Intel nos diz que ela pode falhar se leituras forem feitas em intervalos menores que 10 milissegundos, o que nos colocaria num loop por cerca de uns 5 milhões de ciclos de máquina, mas a coisa não é tão crítica assim... E, mesmo que seja, perder um tempinho (10 ms, no máximo) para garantir a aleatoriedade dos valores é um preço pequeno a pagar.

O outro problema do gerador de congruência de linear que não é nem um pouco óbvio (problema, aliás, que RDRAND não tem!) é que os bits inferiores do valor obtido tendem a ser menos aleatórios que o restante dos bits<sup>63</sup>... Suponha que estamos querendo obter um valor entre 1 e 6 para simular o lançamento de um dado de seis faces. O programador pode escolher fazer algo assim:

```
value = (rand() % 6)+1;
```

Ao obter o resto da divisão por 6 estamos aproveitando apenas os 3 bits inferiores do resultado de *rand()*. Podemos obter mais faces voltadas para o 3, no dado, do que as demais faces! Uma possível solução, usando *rand()*, é obtermos os 4 bits superiores do resultado (já que o bit 31 sempre estará zerado!):

```
value = ((rand() >> 28) % 6) + 1;
```

A solução óbvia, se seu processador possuir a instrução RDRAND, é usar nossa função *getrandom()* ao invés de *rand()*. Mas, fique atento que ao obter o resto da divisão por 6, o valor aleatório obtido ainda é um pouco sofável... O ideal é escolher um divisor múltiplo de  $2^n$ . Isso equivale a isolar bits inferiores do valor através do uso de uma máscara com uma simples instrução AND:

```
value = getrandom() & 0x7f; /* Obtem um valor aleatório entre 0 e 127 */
```

A instrução RDRAND vem em 3 sabores: 16, 32 ou 64 bits, dependendo do registrador usado como operando. Poderíamos escrever o seguinte macro em assembly inline<sup>64</sup>:

```
#define RANDOM(x) \
    __asm__ __volatile__ ( \
        "1:\n" \
        "rdrand %0\n" \
        "jnc 1b" : "=g" ((x)) \
    )
```

O compilador escolherá a variação de RDRAND apropriada dependendo do tamanho da variável passada para o macro...

---

63 Tenho que agradecer a Nelson Brito por me indicar esse detalhe... Num projeto, elaborado pelo Nelson, ele usava um macro com valores em ponto flutuante para lidar com esse problema. No momento em que o entendi pude otimizar algumas rotinas para uso exclusivamente de valores inteiros.

64 Uma explicação sobre o salto condicional no assembly inline: Labels numéricos devem ser referenciados com um sufixo 'b' ou 'f' (de "back" e "forward"). Ao usar 'b' o salto é feito para o label anterior (para trás).

# Capítulo 15: Misturando Java e C

Misturar Java e C é uma coisa muito simples de ser feita. O *Java Development Kit* (JDK) possui bibliotecas e *headers* para que o desenvolvedor possa usar rotinas externas à JVM. O padrão chama-se JNI (*Java Native Interface*).

Eu disse “simples”? Java tem lá suas idiossincrasias... Embora usar a JNI seja simples, é preciso entender como java funciona para usá-la corretamente. O uso incorreto acarreta sérios problemas para a aplicação, já que a JVM (*Java Virtual Machine*) é responsável por gerenciar todo o ambiente do seu próprio jeito, que é incompatível com a maneira como a *libc* lida com os seus recursos.

Meu objetivo em adicionar Java na discussão num livro sobre C e Assembly é mostrar o quanto esquisita é essa linguagem (java!) e te dar um vislumbre de todo o trabalho que a JVM faz por debaixo dos panos. Isso esclarece porque não sou lá muito “chegado” em linguagens em ambientes gerenciados como Java e C#.

## **Porque não é uma boa ideia misturar C com ambientes “gerenciados”**

Por ambientes “gerenciados” quero dizer linguagens como Java e C# e qualquer outra que use o conceito de *Garbage Collection*. Nesses ambientes aplicações são executadas sob o que chamamos “máquinas virtuais”. Têm esse nome porque elas emulam uma máquina fictícia, com regras de gerenciamento de memória e “linguagem de máquina” próprias. Ao usar códigos feitos em C nesses ambientes você poderá usar funções da *libc* para gerenciar memória, como *malloc*, mas a alocação de memória dinâmica não estará sob controle do *Garbage Collector*.

Tudo no Java (e no C#), exceto pelos tipos primitivos (byte, char, short, int, long, float e double) são objetos. E todo objeto é usado através de um ponteiro “disfarçado”. Ponteiros são chamados de “referências” em Java. Mas é um pouco mais complicado do que isso...

Em C++, na definição de uma classe, podemos dizer ao compilador para usar uma tabela de ponteiros para as funções-membro. Essas funções são marcadas como “virtuais”. Mas C++ também permite a declaração de funções-membro não-virtuais. Para ilustrar a diferença, os códigos abaixo são mais ou menos equivalentes:

```
/* MyClass.cc */
class MyClass
{
public:
    int x;

    virtual int getX(void);
};

int MyClass::getX(void) { return this->x; }
-----%<---- corte aqui ---->%<-----
/* MyClass.c */
struct MyClass;

struct MyClass_vtbl
{
    int (*getX)(struct MyClass *this);
};

static MyClass_vtbl myclass_vtbl;
```

```

struct MyClass
{
    int x;

    struct MyClass_vtbl *vtbl;
};

int MyClass_getX(struct MyClass *this) { return this->x; }

void MyClass_ctor(struct MyClass *p)
{
    p->vtbl = &myclass_vtbl;
    p->vtbl->getX = MyClass_getX;
}

```

Em C++ toda chamada a uma função virtual é feita através de ponteiros contidos na tabela virtual, mais ou menos como é descrito no código em C, acima... Um objeto tem, atrelado a si, um ponteiro para uma estrutura contendo os ponteiros para as funções virtuais da classe. Ao chamar a função esse ponteiro é automaticamente usado (em C++). Mais ou menos assim:

```

/* teste.cc */
MyClass obj;
...
int y = obj.getX();
-----%<----- corte aqui -----%<-----
/* teste.c */
struct MyClass obj;

MyClass_ctor(&obj);
...
int y = obj.vtbl->getX(&obj);

```

É fácil perceber que toda chamada a funções virtuais são feitas de forma indireta, pela tabela virtual. Isso existe para que, durante a herança, o polimorfismo possa ser facilmente implementado. Vou deixar esse detalhe de fora por aqui...

No java, por default, todo método (outro nome que java dá às funções-membro) é virtual. E já que usar objetos é sempre feito por referência, usando ponteiros, toda chamada de método é feita por uma indireção dupla... Temos o ponteiro “disfarçado” sob o nome do objeto e temos o ponteiro “escondido” da tabela virtual. Uma comparação superficial de chamadas usando códigos em Java e C, seria assim:

```

// Em java:
obj.f();
-----%<----- corte aqui -----%<-----
/* Em C: */
obj->vtbl->f(); /* obj é um ponteiro! */

```

É necessário entender isso se você for usar a JNI...

## **Garbage Collection, no Java**

Outra coisa que é preciso entender é como Java usa a memória. A JVM pré-aloca e divide o *heap* em, basicamente, 3 blocos. O primeiro bloco é chamado *Eden* ou *Young Generation*, que contém objetos pequenos e de vida curta. O segundo é chamada *Elder Generation*. Ele contém objetos maiores e que conseguiram “sobreviver” ao *Garbage Collector*, vindos da *Young Generation*. Já o terceiro bloco é chamado *Permanent Generation*.

Objetos contidos nas regiões *Eden* e *Elder* são movidas o tempo todo pelo coletor de lixo. Aqueles objetos que não estão mais em uso precisam ser liberados e os que ainda estão sendo usado precisam ser movidos para evitar a fragmentação dessas regiões. Como esses objetos são movidos, seus ponteiros são alterados com frequência. É para manter um registro da nova posição do objeto que existe a região *Permanent*. Ponteiros para os objetos nas gerações sob a influência do *Garbage*

*Collector* são mantidos na região *Permanent* e quando o coletor move algum bloco tudo o que ele tem que fazer é alterar o ponteiro da referência no lado permanente. Assim, o código em Java sabe onde o objeto está o tempo todo:

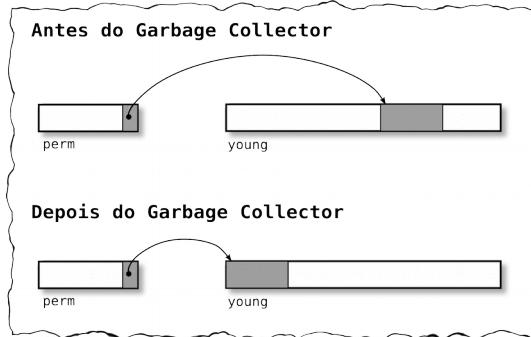


Figura 18: Garbage Collector movimentando um único objeto.

Juntando o conceito das gerações com as funções virtuais, java mantém o ponteiro para as estruturas do objeto na região permanente e esse ponteiro aponta para a tabela virtual numa das regiões sob influência do Garbage Collector e a tabela virtual contém um ponteiro para uma outra geração “escondida” que contém o bytecode da função...

### Misturando com um exemplo simples...

A primeira coisa a ser feita quando for misturar seus códigos em C com o ambiente Java é criar uma classe que carregue uma *shared library*, definida por você, que contenha as suas funções:

```
// HelloWorld.java
//
// Essa classe é um "stub". Ela contém as declarações das funções "nativas".
//
public class HelloWorld {
    // Inicializador carrega a shared library libHello.so.
    static { System.loadLibrary("Hello"); }

    // A função "nativa" em C: void showMessage(void);
    public static native void showMessage();
}
```

O objeto da classe *HelloWorld*, através do inicializador, vai carregar o *shared object* “libHello.so” e ela também declara a função *showMessage*, que se localiza nessa biblioteca... Se o código for executar no Windows, provavelmente a JDK vai procurar por *hello.dll* nos diretórios indicados pela variável de ambiente PATH.

Depois de compilar a classe você poderá usar o utilitário 'javah' para obter o header com as declarações das funções da classe (listada abaixo sob o nome *HelloWorld.h*):

```
$ javac HelloWorld.java
$ javah -o HelloWorld.h HelloWorld
$ ls -l
-rw-r--r-- 1 user user 152 Jan 21 16:07 libHello.c
-rw-r--r-- 1 user user 130 Jan 21 16:16 Hello.java
-rw-r--r-- 1 user user 388 Jan 21 16:29 HelloWorld.h
-rw-r--r-- 1 user user 120 Jan 21 16:04 HelloWorld.java
-rw-r--r-- 1 user user 448 Jan 21 16:58 Makefile
-----%<---- corte aqui ----%<-----
```

```

/* HelloWorld.h (criado por javah) */
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>

/* Header for class HelloWorld */

#ifndef _Included_HelloWorld
#define _Included_HelloWorld

#endif /* cplusplus
extern "C" {
#endif

/*
 * Class:      HelloWorld
 * Method:    showMessage
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloWorld_showMessage
(JNIEnv *, jclass);

#endif /* cplusplus
}
#endif
#endif

```

Eis a rotina em C que comporá a *shared library*. Ela deve incluir o header gerado acima:

```

/* libHello.c */
#include <stdio.h>
#include "HelloWorld.h"

JNIEXPORT void JNICALL Java_HelloWorld_showMessage(JNIEnv *env, jclass obj)
{
    puts("Hello, world!");
}

-----%<---- corte aqui -----%>-----
$ gcc -O3 -I /usr/lib/jvm/default-java/include -march=native -fPIC -o libHello.o libHello.c
$ gcc -o libHello.so -shared libHello.o

```

Importante reparar que o nome da nossa função obedece o nome contido no header. O java usa um esquema de nomenclatura especial (o exemplo acima é só um caso simples – existem nomes mais confusos).

Aqui surge o primeiro problema. É necessário dizer ao GCC onde é que podemos encontrar o header *jni.h*, que acompanha o JDK. A conclusão óbvia que segue desse fato é que sua *shared object* é dependente da versão do JDK usado.

Tudo o que precisamos fazer agora é criar uma classe java que vai instanciar nossa classe *HelloWorld*, que contém as declarações nativas e executar o código:

```

// HelloWorld.java
public class Hello {
    public static void main(String[] args) {
        HelloWorld hw = new HelloWorld();

        hw.showMessage();
    }
}

-----%<---- corte aqui -----%>-----
$ javac -classpath . Hello.java
$ LD_LIBRARY_PATH=. java Hello
Hello, world!

```

Repare que 'javac' precisa saber onde a classe *HelloWorld* está, por isso defini o *classpath* apontando para o diretório corrente. Ainda, 'java' precisa saber onde o nosso *libHello.so* está. Por isso, forcei a barra e defini *LD\_LIBRARY\_PATH* apontando para o diretório corrente. Na prática, você poderá colocar sua *shared library* em */usr/lib*, por exemplo. De novo, no Windows isso não é

necessário: DLLs são procuradas, pelo sistema operacional, nos diretórios contidos na variável de ambiente PATH e no diretório corrente.

Eis um *makefile* para tudo isso:

```
# Makefile
#
# Use: make           (para compilar)
#       make run        (para executar Hello.class)
#       make clean       (para apagar arquivos deixando apenas os fontes)

JDK_LIBPATH=/usr/lib/jvm/default-java/include

.PHONY: all run clean

all: Hello.class HelloWorld.class libHello.so

libHello.so: libHello.o
    gcc -o $@ -shared $^

libHello.o: libHello.c HelloWorld.h
    gcc -I $(JDK_LIBPATH) -c -fPIC -O3 -march=native $@ libHello.c

# javah precisa receber apenas o nome da classe.
HelloWorld.h: HelloWorld.class
    classfile="$<"; \
    javah -o $@ ${classfile%.class}

%.class: %.java
    javac -classpath . $<

run:
    LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH java Hello

clean:
    rm *.o *.so *.class *.h
```

Voltando ao código em C, os tipos usados pelo Java são diferentes. São nomeados com um 'j' na frente:

Tipo “javanês”	Tipo “C”
jboolean	unsigned char
jbyte	char
jchar	unsigned short
jshort	short
jint	int
jlong	long
jfloat	float
jdouble	double
jclass	?
jstring	?
jobject	?
j<T>Array	<T>[]
void	void

Tabela 13: Tipos usados no JNI

Na tabela acima, o <T> é um tipo primitivo ou “object” do java e o tipo equivalente do C. Um array

do tipo `jintArray` é mapeado para 'int []', em C, por exemplo.

Os tipos `jclass`, `jstring` e `jobject` são tipos especiais. Todos os tipos não primitivos são objetos, em Java. Para acessá-los temos que usar algumas funções disponibilizadas pela JVM acessíveis através do ponteiro do tipo `JEnv`, passado para a nossa função.

Aqui entra o conceito de funções virtuais que mostrei anteriormente. Esse parâmetro do tipo `JEnv` é um ponteiro para uma tabela que contém ponteiros para 230 funções disponibilizadas pela JVM. Ou seja, é um ponteiro para uma tabela virtual.

## Usando strings

Além dos tipos primitivos (int, char, float...) todos os outros são objetos, em Java, e são passados para a nossa função como tal. Sendo objetos, esses parâmetros podem ser movidos num dos heaps gerenciados pelo java. Daí, temos que tomar precauções especiais com relação a eles.

Tenho boas notícias e más notícias com relação às strings no Java. Qual você quer ler primeiro? Bem... A boa notícia é que todas as strings, em java, são codificadas no formato UTF-8. Mesmo que você edite o código fonte usando um charset mais simples, como WINDOWS-1252 ou ISO-8859-1, o compilador converterá suas strings para UTF-8.

A má notícia é que java usa um formato proprietário para o `charset` UTF-8. Segundo a documentação da Oracle, por exemplo, o caractere nulo ('\0') não corresponde ao byte zero, mas aos bytes '\xc0\x80'. Isso é um grande problema com relação ao esquema de codificação de strings usado pelo C, que espera um caractere nulo no final. Felizmente, ao que parece, java mantém essa regra, reservando o caractere especial para '\0' **explícito**.

Tenha em mente que alguns caracteres especiais poderão aparecer errados, do ponto de vista de C, graças às idiossincrasias do java.

Para lidar com as strings, ou você as obtém nesse formato e lida com elas assim mesmo, ou as converte para *unicode*. Unicode, no contexto de C, é o tipo `wchar_t`. Cada caractere terá tamanho fixo de 16 bits. A vantagem de usar UTF-8 é que ele tem mapeamento direto ao ASCII. A desvantagem é que este é um *multibyte charset*. Isso quer dizer que alguns caracteres podem ter mais que um byte de tamanho. No Linux isso não é problema, já que o charset default é o UTF-8. No Windows, isso pode ser meio chato... Windows usa, internamente, o formato unicode *single charset*, onde cada caractere tem 2 bytes de tamanho. Em C, se for declarar uma string literal usando unicode, deve-se usar um 'L' na frente da string:

```
wchar_t mystring[] = L"Teste de string em unicode.;"
```

Neste caso, para evitar avisos do compilador, você deverá fazer conversões de tipos no seu código, ao chamar uma das funções da JVM:

```
jstring s = (*env)->NewString(env, (const jchar *)mystring, wcslen(mystring));
```

Ou fazer o *type casting* para "const `wchar_t`\*". Claro que esse *casting* pode não ser necessário. O tipo `jchar` é definido como *short* e, portanto, tem 16 bits de tamanho.

Para obter acesso ao conteúdo do objeto String temos que obter uma referência, travá-la para que o *Garbage Collector* não bagunce tudo movendo os dados de um lugar para outro e, depois de usar os dados do buffer apontado pela referência, liberar a trava. Para tanto, existem pares de rotinas que devem sempre ser usadas para lidar com strings: `GetString*` e `ReleaseString*`. A seguir, eis um exemplo de uso:

Suponha que a nossa função `showMessage`, no exemplo anterior em java, seja assim:

```
public static native void showMessage(String str);
```

Ao executar *javah* em *HelloWorld.class*, obtemos um arquivo *HelloWorld.h* com um protótipo e mapeamos esse protótipo para a função abaixo:

```
JNIEXPORT void JNICALL Java_HelloWorld_showMessage(JNIEnv *env, jclass this, jstring str)
{
    const char *s;

    /* Obtém o ponteiro da string em UTF-8. */
    if ((s = (*env)->GetStringUTFChars(env, str, NULL)) != NULL)
    {
        puts(s);

        /* Liberamos o ponteiro da string em UTF-8. */
        (*env)->ReleaseStringUTFChars(env, str, s);
    }
}
```

Usamos 'env' para chamar as funções da JVM. Note que o ponteiro 'env' contém um ponteiro que aponta para uma tabela virtual. Por isso, em C, temos que usá-lo como '(\*env)'. Se estivéssemos usando C++ e o tipo *JEnv* fosse declarado como uma classe contendo funções virtuais (existe uma em *jni.h*), então poderíamos fazer a chamada assim:

```
env->GetStringUTFChars(str, NULL);
```

O ponteiro 'this' e a indireção da tabela virtual é assumida por default, em C++, quando há declaração de chamadas de funções de classes, que sejam virtuais... Mas, estamos lidando com C aqui...

No exemplo, *GetStringUTFChars* obtém um ponteiro para o buffer contendo a string passada como parâmetro. O terceiro parâmetro, NULL, passado para *GetStringUTFChars*, diz à JVM que ela não tem que fazer uma cópia da string original. Este parâmetro é um ponteiro para o tipo *jboolean* que, se apontar para um valor *JNI\_TRUE*, indica para a JVM que vamos trabalhar com uma cópia, não com o buffer original:

```
jboolean copy = JNI_TRUE;
s = (*env)->GetStringUTFChars(env, str, &copy);
```

Cópias podem ser alteradas o quanto quisermos. Trabalhar com o buffer original não é lá uma boa ideia, já que ele deveria ser gerenciado apenas pela JVM. Nada impede que o meu próprio código crie uma cópia da string original. Por isso costumo passar NULL ou zero todas as vezes:

```
const char *s;
char *s2;

s = (*env)->GetStringUTFChars(env, str, NULL);
s2 = strdup(s);
...
free(s2);
(*env)->ReleaseStringUTFChars(env, str, s);
```

É bom verificar o resultado da chamada a *GetString\**. Ela pode retornar um ponteiro NULL, nos dizendo que houve um erro na JVM (*OutOfMemory?*).

A chamada a *ReleaseStringUTFChars* é necessária para fazer o destravamento do objeto.

Agora, suponha que a função, em java, devolva uma string:

```

...
public static native String getMessage();
...
-----%<---- corte aqui ----%<-----
static const char myString[] = "Teste";

JNIEXPORT jstring JNICALL Java_HelloWorld_getMessage(JNIEnv *env, jclass this)
{
    return (*env)->NewStringUTF(env, myString);
}

```

A função *NewStringUTF* criará um objeto *String* em algum lugar do heap da JVM (possivelmente na geração *Young*), obterá a string do buffer apontado por *myString* (criando a sua própria cópia) e devolverá a referência dada pelo tipo *jstring* para a JVM. Mas, é garantido que essa referência seja uma referência *local*, do ponto de vista do java... Isso significa que, se nenhuma função fora da nossa for usar a string, é responsabilidade da nossa função indicar à JVM que essa referência poderá ser coletada e colocada no lixo...

No exemplo acima, retornamos a referência (o tipo *jstring* é um ponteiro!) para o método chamador e deixamos que ele lide com a referência. Mais adiante mostrarei um caso onde teremos que lidar com a referência por nós mesmos...

## **Usando arrays unidimensionais**

Arrays também são objetos e comportam-se de forma muito parecida com strings. Pelo menos para leitura.

O tamanho de uma string, tanto em C quanto em Java, pode ser conhecido pelo caractere '\0', no final do buffer. No caso de arrays, o tamanho ou é conhecido de antemão, ou pode ser obtido através de uma chamada à função *GetArrayLength* da JVM.

A leitura de um array pode ser feita usando *Get<T>ArrayElements* e *Release<T>ArrayElements*, onde <T> corresponde a um tipo primitivo ou “Object”:

```

int *parray;

/* Pega o array... */
parray = (*env)->GetIntArrayElements(env, javaArrayVar, NULL);
if (parray)
{
    /* Manipula o array, em C, aqui...
     ...

    /* Finalmente, libera o array */
    (*env)->ReleaseIntArrayElements(env, javaArray, parray, 0);
}

```

*Get<T>ArrayElements* funciona igualzinho a *GetString*. Já *Release<T>ArrayElements* tem um parâmetro adicional que especifica o que deve ser feito do buffer. Ele aceita 3 valores:

Modo	Significado
0	Copia de volta para o array java e libera o buffer.
JNI_COMMIT	Copia de volta e não libera o buffer.
JNI_ABORT	Não copia de volta.

Tabela 14: Modos aceitos por *Release<T>ArrayElements*.

Para criar novos arrays existem as funções *New<T>Array* que tomam o número de elementos como parâmetro. Essa função somente aloca espaço na JVM para um array. Para preenchê-lo precisamos usar a função *Set<T>ArrayRegion*, que toma o objeto da classe do array devolvido por

`New<T>Array`, o índice inicial, a quantidade de elementos e o ponteiro para o nosso array, em C. Isso copiará nosso array para o espaço previamente alocado pela JVM:

```
int myArray[3] = { 1, 2, 3 };
jintArray javaArray;

if ((javaArray = (*env)->NewIntArray(env, 3)) != NULL)
{
    (*env)->SetIntArrayRegion(env, javaArray, 0, 3, myArray);
    ...
}
```

A coisa começa a complicar se tivermos arrays com mais de uma dimensão. Para compreender essa complicação precisamos, primeiro, entender como lidar com o tipo *object*...

## Usando objetos

O tipo *object* é uma espécie de coringa no baralho dos tipos do java. Com esse tipo podemos usar qualquer tipo de **referências** a outras classes de armazenamento, incluindo arrays, classes, objetos, tipos primitivos, funções e interfaces. É como se *object* fosse um ponteiro *void*, em C.

Você deve ter reparado que nas funções nativas o segundo parâmetro é do tipo *jclass*<sup>65</sup>. Isso acontece porque podemos querer saber de qual objeto a função foi chamada. Então nossa função nativa recebe a referência *this* por esse parâmetro.

Tome cuidado que, embora esse parâmetro seja declarado como *jclass*, ele não recebe uma referência para uma classe java. Ele recebe a referência para a instância do objeto a qual a função native pertence.

Para mim isso sempre foi meio estranho: Em java, classes e objetos são, ambos, “objetos”!

Java tem objetos dos mais variadas. Até mesmo arrays são objetos. E para lidar com objetos precisamos saber qual é a sua classe. Por exemplo, se quiséssemos alocar espaço para um objeto da classe “*MyClass*”, vazio, teríamos que fazer algo assim:

```
jclass oclass = (*env)->FindClass(env, "LMyClass;");
jobject obj = (*env)->AllocObject(env, oclass);
```

A string passada para a função *FindClass* é uma **assinatura** que descreve a classe de armazenamento. É uma forma abreviada de descrever o objeto e a tabela abaixo mostra os descritores usados.

---

<sup>65</sup> Pode ser substituído, sem medo, pelo tipo *jobject*, que faria mais sentido. Mas esse é o protótipo que *javah* gera...

Classe	Assinatura	Significado
byte	B	
char	C	
double	D	
float	F	
int	I	
long	J	
short	S	
void	V	
boolean	Z	
$T[]$	[T]	Array to tipo $T$
Classe de objeto	Lclasspath;	A classe <i>String</i> , por exemplo, é descrita como “Ljava/lang/String;” (os pontos são substituídos por '/').
Método	(args)T	'args' é uma lista delimitada por ';' e T é o tipo de retorno.

Tabela 15: Lista de assinaturas que descrevem tipos, em Java.

Nossa classe *Hello*, por exemplo, possui as seguintes assinaturas:

```
$ javap -s Hello
public class Hello {
    public Hello();
        Signature: ()V

    public static void main(java.lang.String[]);
        Signature: ([Ljava/lang/String;)V
}
```

O utilitário *javap*, que acompanha o JDK, é um pré-processador e disassembler. Ele te permite ver o bytecode gerado pelo compilador *javac*, bem como outras informações... A opção '-s' lista apenas as assinaturas (signatures) da classe.

No caso de *main* a assinatura “[Ljava/lang/String;)V” nos diz que este objeto é uma função que retorna void (os parênteses e o 'V' no final) e possui um único argumento que é um array para uma string, “[Ljava/lang/String;”.

Suponha que tenhamos uma classe com um membro de dados e um método assim:

```
private String[] slist;
public int[][] getArray(int size, String str);
```

As assinaturas serão, respectivamente: “[Ljava.lang.String;” e “(I;Ljava.lang.String;)[[I”.

Mesmo funções são objetos no Java, por isso possuem assinaturas descriptivas.

### Chamando métodos do próprio objeto

Se nossa função nativa precisar chamar um método do próprio objeto onde foi declarada, podemos fazer algo assim:

```

// HelloWorld.java
public class HelloWorld {
    static { System.loadLibrary("Hello"); }

    public static native void call();

    private void callback() { System.out.println("chamada por C."); }
}

-----%----- corte aqui -----%-----

/*
 * definida como "public static void call();"
 * na classe HelloWorld. */
JNIEXPORT void JNICALL Java_HelloWorld_call(JNIEnv *env, jclass this)
{
    jclass cls;
    jmethodID mid;

    /* Obtém a classe da referência ao objeto 'this'. */
    cls = (*env)->GetObjectClass(env, this);

    /* Obtém o índice da tabela virtual da classe.
     * Se retornar NULL significa que não achou o método.
     */
    if ((mid = (*env)->GetMethodID(env, cls, "callback", "()V")) != NULL)
    {
        /* Executa o método da entrada nº 'mid' da tabela virtual do objeto. */
        (*env)->CallVoidMethod(env, this, mid);
    }
}

```

O fato de que *GetMethodID* precisa usar o nome da função para obter sua posição na tabela virtual nos diz que o código compilado via JIT (*Just In Time compiler*), que supostamente gera o código em assembly equivalente ao bytecode, vai usar, necessariamente *RTTI* (*RunTime Type Information*). Isso quer dizer que o código final **nunca** será tão performático quanto um código em C ou assembly, já que haverá muita comparação com strings!

No caso da função a ser chamada precisar de parâmetros, precisamos prepará-los. E a função *Call<T>Method* aceita múltiplos parâmetros, do mesmo jeito que funções como “printf” o fazem:

```

/ HelloWorld.java
public class HelloWorld {
    static { System.loadLibrary("Hello"); }

    // membro de dados... usado mais adiante.
    public int myField;

    public static native void call();

    private void callback(String str) { System.out.println(str); }
}

-----%----- corte aqui -----%-----

JNIEXPORT void JNICALL Java_HelloWorld_callback(JNIEnv *env, jclass this)
{
    static char s[] = "Chamada pelo código nativo!";
    jstring str;
    jclass cls;
    jmethodID mid;

    if ((str = (*env)->NewStringUTF(env, s)) != NULL)
    {
        /* Obtém a classe da referência do objeto 'this'. */
        cls = (*env)->GetObjectClass(env, this);

        /* Obtém o índice da tabela virtual da classe.
         * Se retornar NULL significa que não achou o método.
         */
        if ((mid = (*env)->GetMethodID(env, cls, "callback", "()V")) != NULL)
        {
            /* Executa o método da entrada nº 'mid', do objeto passando um parâmetro adicional. */
            (*env)->CallVoidMethod(env, this, mid, str);
        }
    }
}

```

```

    /* Livra-se da referência. */
    (*env)->DeleteLocalRef(env, str);
}
}

```

A string passada para o “callback” precisa ser criada e sua referência é sempre local. Já que não vamos devolvê-la para alguma função chamadora, temos que marcá-la como “descartável”, usando a função *DeleteLocalRef*. Sem isso teríamos um *memory leak*.

## Acessando membros de dados do próprio objeto

Acessar um membro de dados é feito do mesmo jeito que fizemos para conhecer o ID do método. Só que o ID obtido é do membro de dados, chamado no java de *field*:

```

jclass cls = (*env)->GetObjectClass(env, this);
jfieldID fieldID = (*env)->GetFieldID(env, cls, "myfield", "I");
int myIntField;

if (fieldID != NULL)
{
    myIntField = (*env)->GetIntField(env, this, fieldID);
    /* ... faz algo com 'myField' aqui. ... */
}

```

Não há necessidade de nos livrarmos da referência, já que ela pertence ao próprio objeto.

Note que, ao obter o *FieldID*, temos que passar, além do nome do campo, a assinatura do mesmo. De novo, java parece ser fortemente ligado ao *RTTI*.

*GetFieldID* pode retornar NULL, indicando que o membro de dados não pôde ser encontrado no objeto. Omiti a verificação de erros por pura preguiça...

## De volta aos arrays: Usando mais de uma dimensão

Arrays com mais de uma dimensão são, na verdade, arrays de arrays. O primeiro nível é sempre um array de *objects*. Somente a última dimensão é que é um array de um tipo primitivo. Ao declarar uma função nativa:

```

// Em Java:
public static native int[][] getBidimensionalArray();

```

O que a função, em C, de fato retorna é uma referência a um array do tipo *jobjectArray*.

Para criar esse array no seu código você usará *NewObjectArray*, obterá os ponteiros com *GetObjectArrayElements*, para cada elemento do array usará *NewIntArray* (como descrito anteriormente) e usará *SetIntArrayRegion* para preenchê-los.

Já que o que estamos retornando ao chamador é uma referência a *jobjectArray*, você deverá informar à JVM, depois de liberar o array de inteiros com *ReleaseIntArrayElements*, que esses devem ser marcados como “descartáveis”, usando *DeleteLocalRef*.

O método chamador lidará com a referência ao array de *objects*. Mas os arrays mais internos deverão estar marcados como descartáveis para não causar um *memory leak*, já que o método chamador não foi o responsável pela criação destes!

O acesso a arrays de múltiplas dimensões é feito de maneira semelhante:

```

// Em Java:
public static native int doSomething(int[][] array);

```

A nossa função nativa recebe um array de *objects* e você deve obter os arrays mais internos a partir

dos elementos do array mais externo (de *objects*).

Lembre-se que arrays são objetos em si. Eles têm funções membro atreladas. Uma delas é *GetArrayLength*, que devolve o número de elementos do array. Essa função pode ser usada para conhecer o tamanho das dimensões:

```
/* Pega elementos do primeiro nível. */
int obj_elements = (*env)->GetArrayLength(env, array);
int i;

for (i = 0; i < obj_elements; i++)
{
    /* Pega elementos do segundo nível. */
    jobject o = (*env)->GetObjectArrayElement(env, array, i);
    int *iarr = (*env)->GetIntArrayElements(env, o, 0);
    int iarr_elements = (*env)->GetArrayLength(env, o);
    int j;

    for (j = 0; j < iarr_elements; i++)
    {
        /* ... faz algo com os elementos dos array inteiro (ponteiro iarr). ... */
    }

    (*env)->ReleaseIntArrayElements(env, o, iarr, 0);
}
...
```

E, já que recebemos o array do chamador, não somos responsáveis por liberar quaisquer referências locais.



# Capítulo 16: Usando Python como script engine

Também não gosto muito de linguagens interpretadas, as chamadas “script languages”. Elas tem que ser “interpretadas” (parsed), quando o “parser” busca por erros na gramática e depois “executadas” passo por passo. É claro que a performance fica seriamente comprometida no processo. Mas, elas são úteis...

Imagine um cenário onde você possa criar algumas rotinas, em modo texto, que possam ser chamadas pelo seu código, em C, mais performático. Nesse caso, os “scripts” podem ser modificados quando necessário sem afetar muito o seu código original.

Dentre as linguagens “script” mais badaladas, o Python tem uma vantagem: Ele pode ser pré-compilado. Além de ter uma série de recursos interessantes, em sua linguagem.

## Contagem de referências

Assim como Java, o interpretador Python também tem um *garbage collector*. Ele funciona de uma maneira muito simples: Todo objeto tem um contador interno que começa em 1 quando é instanciado e esse é contado é incrementado a cada vez que é criada uma nova referência ao objeto, por exemplo, se ele é passado como parâmetro para uma função. Sempre que essa referência sai do escopo, o contador é decrementado. Quando o contador chega a zero, o “*garbage collector*” livra-se do objeto.

O interpretador faz isso automaticamente, mas nosso código, ao usar objetos do python, não tem esse luxo. Temos que usar os macros *Py\_INCREF* e *Py\_DECREF*. Raramente precisaremos usar *Py\_INCREF*, mas devemos usar *Py\_DECREF* se pretendermos marcar o objeto como “descartável”, assim como fazemos no java, com a função *DeleteLocalRef*.

*Py\_DECREF* tem um problema. Se a referência chega a zero o objeto é liberado e também é setado para NULL. E usar ponteiros nulos causa *segmentation faults*. Isso pode ser resolvido com o uso de *Py\_XDECREF*, que faz a mesma coisa que o macro anterior, mas ignora referências nulas... O problema é que, ao usar *Py\_XDECREF*, não saberemos se nosso programa tem um bug ou não. É bom restringir seu uso para os casos onde você sabe que a referência **pode** ser NULL.

## Instanciando o python

Abaixo temos um esqueleto de programa que usa o interpretador python. Nosso nosso programa hospedeiro (*host*):

```
#include <Python.h>

int main(int argc, char *argv[])
{
    Py_Initialize();
    if (!Py_IsInitialized())
    {
        fprintf(stderr, "Não foi possível carregar o Python.\n");
        return 1;
    }

    /* Usar as funções da libpythonX.X aqui. */
    Py_Finalize();
    return 0;
}
```

É bem simples, huh?

## Carregando um módulo

Assim como na linguagem C, um arquivo com extensão '.py', que contém código python, é chamado de “módulo”. Precisamos saber como carregar um módulo contendo nossas funções e classes. É bem simples, mas tem um detalhe.

Python carrega seus módulos a partir de um conjunto de diretórios pré-definidos. Podemos saber quais são com esse código:

```
#!/usr/bin/python
# showsyspath.py
import sys

print sys.path
-----%<---- corte aqui -----%<-----
$ ./showsyspath.py
['/home/user/work/book-srcs/python', '/usr/lib/python2.7',
 '/usr/lib/python2.7/plat-x86_64-linux-gnu', '/usr/lib/python2.7/lib-tk',
 '/usr/lib/python2.7/lib-old', '/usr/lib/python2.7/lib-dynload',
 '/usr/local/lib/python2.7/dist-packages', '/usr/lib/python2.7/dist-packages',
 '/usr/lib/python2.7/dist-packages/PILcompat', '/usr/lib/python2.7/dist-packages/gtk-2.0',
 '/usr/lib/pymodules/python2.7', '/usr/lib/python2.7/dist-packages/ubuntu-sso-client']
```

O primeiro diretório de busca dos módulos é o diretório atual. Mas isso não funciona bem com módulos carregados a partir de nossos programas em C. Ao inicializar o interpretador com *Py\_Initialize*, a lista dos diretórios estará em *sys.path* **exceto** o diretório atual. Por isso, temos que incluir o diretório onde o módulo se encontra, se não for um dos diretórios default.

A listagem seguinte mostra como o módulo *mymodule.py* é carregado a partir do diretório onde nosso programa será chamado:

```
#include <Python.h>

#define MODULE_NAME "mymodule"

void executeModule(Py_Object *);

int main(int argc, char *argv[])
{
    Py_Object *sys, *path;
    Py_Module *module;

    Py_Initialize();
    if (!Py_IsInitialized())
    {
        fprintf(stderr, "Não foi possível carregar o Python.\n");
        return 1;
    }

    /* Importa o módulo 'sys' */
    if ((sys = PyImport_ImportModule("sys")) != NULL)
    {
        /* 'sys' é um objeto que contém o atributo 'path' */
        if ((path = PyObject_GetAttrString(sys, "path")) == NULL)
        {
            fprintf(stderr, "Erro ao obter o atributo 'sys.path'.\n");
            goto endOfProgram;
        }

        /* Adiciona o diretório corrente em 'sys.path' */
        /* PyList_Append() "captura" a referência do objeto! */
        PyList_Append(path, PyString_FromString("."));
    }
}

endOfProgram:
    /* Limpa os recursos */
    Py_Finalize();
}
```

```

module = PyImport_ImportModule(MODULE_NAME);
if (module != NULL)
{
    /* Usar as funções da libpythonX.X aqui. */
    executeModule(module);

    /* Não precisamos mais do módulo. */
    Py_DECREF(module);
}

endOfProgram:
    Py_DECREF(sys);
}
else
{
    fprintf(stderr, "Erro ao carregar o módulo 'sys'.\n");
    Py_Finalize();
    return 1;
}

Py_Finalize();
return 0;
}

```

É bem verdade que ao chamarmos *Py\_Finalize*, todas as referências serão liberadas, já que o interpretador também será descarregado. O código acima é literal com relação ao decrementar das referências e, para isso, faço uso de um *goto* e de um label... Considerados como crias do inferno por alguns puristas...

A única coisa estranha é que não precisamos liberar o objeto 'path'. O motivo é que objetos containers, do tipo listas, tuplas, pilha, filas, “capturam” a referência de um objeto para si. Quando o container “some” devido ao decremento de referências, leva consigo seus itens.

## **Executando um módulo simples**

Separei a execução do módulo na função *executeModule*, acima, para facilitar a leitura do código e não ser repetitivo. Para executar uma função simples, em python, temos apenas que chamar a função *PyObject\_CallObject*, passando o objeto a ser chamado (a função) e seus parâmetros. Suponha que tenhamos o seguinte módulo em python:

```

# mymodule.py
def multiply(a,b):
    print "Multiplicando",a,"por",b
    return a*b

```

Nossa função *executeModule* ficaria:

```

void executeModule(PyObject *pModule)
{
    PyObject *pFunc, *pArgs, *pValue;

    /* Obtem o objeto da função e verifica se o objeto é, de fato,
     * uma função. */
    pFunc = PyObject_GetAttrString(pModule, "multiply");
    if (pFunc != NULL && PyCallable_Check(pFunc))
    {
        /* Prepara 2 parâmetros para serem passados para a função. */
        if ((pArgs = PyTuple_New(2)) != NULL)
        {
            /* Os parametros 3 e 2 são colocados na posição 0 e 1 de uma 'tupla'. */
            PyTuple_SetItem(pArgs, 0, PyObject_FromLong(3));
            PyTuple_SetItem(pArgs, 1, PyObject_FromLong(2));

            /* Finalmente, chama a função. */
            pValue = PyObject_CallObject(pFunc, pArgs);
        }
    }
}

```

```
/* Deixa o nosso código apresentar o resultado. */
printf("Valor retornado do python: %ld.\n", PyInt_AsLong(pValue));

/* Livra-se das referências */
Py_DECREF(pValue);
Py_DECREF(pArgs);
}

Py_DECREF(pFunc);
}
}
```

# Apêndice A: System calls

A tabela de *system calls*, abaixo, é usada pela instrução SYSCALL do processador, nos sistemas Linux para a arquitetura x86-64 (amd64). Repare que todas as *system calls* estão disponíveis em funções da *libc*.

Ao usar SYSCALL você deve passar os parâmetros em registradores, ao invés da pilha. A documentação do ponto de entrada de uma *syscall*, no código fonte do kernel, nos diz:

Registrador	Significado
RAX	Número da função da syscall, de acordo com a tabela abaixo...
R11	RFLAGS é guardado em R11 antes (se necessário) e no retorno da instrução SYSCALL.
RDI, RSI, RDX, R10, R8, R9	Do primeiro ao sexto parâmetro da função.

Tabela 16: Uso dos registradores numa syscall.

Note que, a não ser pelo registrador RAX, R10 e R11, a instuição SYSCALL segue a convenção de chamada da linguagem C para o modo x86-64. Assim, executar a função *exit* fica assim:

```
mov eax,60          ; função exit.  
mov edi,EXIT_CODE  
syscall            ; chama exit(EXIT_CODE);
```

Outro exemplo. Para escrever uma string na tela, em C, usando a função *write*, precisamos passar o *file descriptor* o FILENO\_STDOUT, o ponteiro para a string e o tamanho dela:

```
char *msg = "Hello, world!\n";  
write(FILENO_STDOUT, msg, strlen(msg));
```

Este descritor tem o valor 1. Assim, a chamada em assembly, usando SYSCALL fica:

```
bits 64  
section .data  
msg:    db  'Hello, world',0x0a  
LENGTH equ $ - msg  
FILENO_STDOUT equ 1  
  
section .text  
...  
; write tem o protótipo:  
; ssize_t write(int fd, void *buffer, size_t count);  
mov eax,1          ; syscall: write  
mov edi,FILENO_STDOUT  
mov rsi,msg  
mov edx,LENGTH  
syscall           ; write(FILENO_STDOUT, msg, LENGTH);  
...
```

## Considerações sobre o uso da instrução SYSCALL

Lembre-se que SYSCALL é usada como interface entre o *userspace* e o *kernelspace*. Todos os ponteiros passados através da convenção de chamada e os ponteiros recebidos no retorno da função (via registrador RAX) são endereços lineares válidos para uso no *userspace*. É o caso, por exemplo, de mmap:

```

; Equivalente a fazer:
;   if ((p = mmap(NULL, 8192, PROT_READ | PROT_WRITE, MAP_ANONYMOUS, -1, 0)) == NULL)
;   {
;     ... error ...
;   }
;
mov eax,9      ; mmap syscall
xor edi,edi    ; addr = NULL
mov esi,8192   ; length = 8192
mov edx,3      ; prot = PROT_READ | PROT_WRITE
mov r10d,32    ; flags = MAP_ANONYMOUS
mov r8d,-1     ; fd = -1
xor r9,r9      ; offset = 0
syscall

mov [p],rax    ; RAX contém o endereço linear do ponteiro de retorno de mmap.
or  rax,rax    ; RAX é NULL?
jnz .L1        ; ... se não for, salta a rotina de erro.
... error ...
.L1:
...

```

Outra consideração importante é a de que seu processador pode não suportar as instruções SYSCALL/SYSRET e as versões anteriores, SYSENTER/SYSEXIT (não use essas!). Para verificar se seu processador suporta este modo basta verificar, via CPUID:

```

mov eax,1
cpuid
test edx,0x800    ; Testa o bit 11 de EDX.
; Neste ponto, se ZF=0, SYSCALL é suportada.

```

Quando o processador não suporta SYSCALL, temos que usar um método que emula as *syscalls*, do mesmo jeito que o modo i386 faz. Neste caso o Linux x86-64 suporta *syscalls* via interrupção 0x80. Os registradores que são usados como parâmetros serão EAX, RBX, RCX, RDX, RSI, RDI e RBP, nessa ordem. Chamamos “int 0x80” e o valor de retorno é colocado em RAX.

Esse acesso às *syscalls* é uma emulação do que é feito no modo i386. A diferença é que ponteiros têm que ser informados nos registradores estendidos de 64 bits. Outros tipos de parâmetros seguem as mesmas regras de tipos para o uso de registradores.

Deve-se levar em conta que os números dos serviços, colocados em EAX, são **diferentes** dos usados com a instrução SYSCALL. Por exemplo, a rotina que imprime uma string ficaria assim:

```

bits 64
section .data
msg:    db 'Hello, world',0xa
LENGTH equ $ - msg
FILENO_STDOUT equ 1

section .text
...
; write tem o protótipo:
; ssize_t write(int fd, void *buffer, size_t count);
mov eax,4          ; syscall: write (modo emulado)
mov edi,FILENO_STDOUT
mov rsi,msg
mov edx,LENGTH
int 0x80           ; write(FILENO_STDOUT, msg, LENGTH);
...

```

Repare que o número da função *syscall\_write* é 4. Quando usamos SYSCALL ele é 1. A mesma coisa acontece com *syscall\_exit*. Usando SYSCALL a função tem número 60, com a “int 0x80” o valor é 1.

## Onde obter a lista de syscalls do Linux?

No código fonte do kernel do Linux você pode obter uma lista de syscalls em

*arch/x86/entry/syscalls*. Existem dois arquivos: *syscalls\_32.tbl* e *syscalls\_64.tbl*, para chamadas via “int 0x80” e SYSCALL, respectivamente. Mais informações sobre uma syscall em particular pode ser obtida na manpage *syscalls* ou nas manpages da função da libc correspondente.

Novamente, a convenção de chamada para a instrução SYSCALL é a mesma do POSIX x86-64 ABI, exceto pelo registrador R10 e R11 (RFLAGS é retornado em R11!).

### ***Usar syscalls no Windows não é uma boa idéia!***

Assim como no Linux, o Windows também tem funções disponibilizadas pelo kernel para uso no *userspace*. Na Win64 API a instrução SYSCALL também pode ser usada, mas o conjunto de registradores de entrada e de saída são diferentes. Da mesma forma, o Windows oferece uma interrupção de software para o caso da instrução SYSCALL não estar disponível, é a interrupção 0x2E (pelo menos as documentadas!).

O grande problema é que as funções não são padronizadas sequer entre versões diferentes do Windows. O Windows NT tem funções numeradas de um jeito, o 2000 de outro, o XP é diferente dos outros dois e assim é para o Vista, 8 e a atual versão 10...

Outro problema é que as *syscalls* do Windows são complicadas de usar. Ao que parece, a Microsoft segue o mesmo padrão usado em rotinas do MS-DOS e BIOS: Ou seja, cada serviço tem suas regras particulares.

Desafio ao leitor a achar algum material bom sobre *syscalls* do Windows na Internet...



# Apêndice B: Desenvolvendo para Windows usando GCC

Uma boa ferramenta para desenvolvimento em C para Windows é o projeto *MinGW* (*Minimalist GNU for Windows*). Com ele você pode usar o *gcc* e o *g++* e algumas ferramentas como *objdump*, *strip*, *ar* e *gas*. A outra vantagem em usar *MinGW* é que você pode desenvolver para Windows sem sair do Linux...

## Usando o MinGW no Linux

Existem dois sabores do projeto: *MinGW32* e *MinGW64*. O procedimento não pode ser mais simples: Instale os pacotes *mingw32* e *mingw64*:

```
$ sudo apt-get install mingw32 mingw64
```

Com isso os compiladores e utilitários (linker, assembler, archiver etc) serão instalados e nomeados com os prefixos “*i586-mingw32msvc-*” e “*x86\_64-w64-mingw32-*”. Eis um simples exemplo de código compilado com o MinGW64:

```
/* hello.c */
#include <windows.h>

int CALLBACK WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpszCmdLine,
                     int nCmdShow)
{
    MessageBox(NULL, "Hello, world!", "Hello", MB_ICONINFORMATION | MB_OK);
    return 0;
}
-----%<---- corte aqui -----%>-----
$ x86_64-w64-mingw32-gcc -O3 -march=native -s -o test.exe test.c -Wl,--subsystem,windows
```

A opção ‘--subsystem,windows’ tem que ser repassada para o linker, senão você acaba com uma aplicação para DOS.

## Usando o MinGW no Windows

Baixe o MinGW de sua preferência (32 ou 64) de um desses sites:

- **MinGW32:** <http://www.mingw.org/>
- **MinGW64:** <http://mingw-w64.sourceforge.net/>

Para instalar qualquer um dos dois o procedimento é, também, bem simples. Basta baixar o instalador e executá-lo. O programa criará uma árvore de diretórios, por exemplo, em *C:\MinGW* e tudo o que você terá que fazer é colocar o subdiretório *C:\MinGW\bin* na variável de ambiente PATH.

A partir daí poderá usar o *gcc* e outros utilitários.

Atualmente o MinGW32 tem um instalador ao estilo *apt-get*, mas gráfico. É bem útil na seleção de “pacotes” do MinGW...

## As bibliotecas *mingwm10.dll* e *msvcrt.dll*

A Microsoft disponibiliza, no diretório *C:\windows\system32*, uma biblioteca dinâmica chmada *msvcrt.dll* (*Microsoft Visual C++ RunTime*) que contém “ganchos” para chamadas à API do Windows. Essa biblioteca funciona como se fosse a *libc*.

*MinGW* precisa de outra biblioteca chamada *mingwm10.dll*, que é uma *surrogate library* para a *msvcrt.dll*. Sem ela seu executável, compilado com o *gcc*, provavelmente não funcionará. A DLL *mingwm10.dll* é distribuída junto com o *gcc* e encontra-se no subdiretório *bin*, no caso do Windows, ou em */usr/share/doc/mingw32-runtime/*, no caso do Linux. Este último disponibiliza a DLL em formato compactado com o *gzip*, portanto, é necessário que você o descompacte antes de copiá-lo para o Windows:

```
$ gunzip /usr/share/doc/mingw32-runtime/mingwm10.dll.gz  
$ cp test.exe mingwm10.dll ~/My_Windows_Virtual_Machine_Shared_Directory/
```

Aparentemente isso não é necessário se você está lidando com o *MinGW64*...

## Limitações do *MinGW*

Como era de se esperar, o *MinGW* não distribui a maioria das *shared libraries* disponíveis no Linux. Se você pretende usá-lo como um *cross compiler*, na tentativa de criar um código único para os dois ambientes (Linux e Windows), esteja avisado: Isso não será fácil!

O *MinGW* do Linux disponibiliza cerca de 150 bibliotecas estáticas relacionadas à API do Windows. Essas bibliotecas são estáticas porque, na verdade, são *surrogates* para *msvcrt.dll* e outras DLLs da API do Windows. No entanto, no site, você pode encontrar algumas bibliotecas adicionais como *zlib*, *pthreads*, *iconv* e *GMP*.

A *libc* usada no *MinGW* também não é completa. Por exemplo, ela não contém a função *asprintf*.

E, finalmente, lembre-se que Windows usa uma convenção diferente, na arquitetura x86-64, no que concerne o tamannho de tipos inteiros longos. Windows usa o padrão IL32P64. Ou seja, 'int' e 'long' têm o mesmo tamanho. Se quiser usar inteiros de 64 bits terá que, obrigatóriamente, usar o tipo 'long long' ou um dos *typedefs* definidos em *stdint.h*.

## Assembly com o *MinGW* e *NASM*

Lembre-se que a convenção de chamada usada no Win64 é diferente da usada no padrão POSIX! Além de se lembrar disso, pouca coisa muda com relação ao *NASM*. Apenas o formato do arquivo objeto terá que ser informado como “win64” ao invés de “elf64” e, no caso da declaração de exportação de símbolos (via diretiva *GLOBAL*), os atributos devem ser omitidos.

## Windows usa codificação de caracteres de 16 bits, internamente

Diferente do Linux, que usa UTF-8, por default, o Windows usa um *charset* próprio onde cada caracter tem 2 bytes de tamanho. Em C isso não é problema, já que podemos escrever uma string desse tipo assim:

```
wchar_t str[] = L"minha string";
```

Dessa forma, cada caracter da string ocupa 2 bytes ao invés de um só:

```
00: 6d 00 69 00 6e 00 68 00 61 00 20 00 73 00 74 00 |m.i.n.h.a. .s.t.|  
16: 72 00 69 00 6e 00 67 00 00 00 |r.i.g...|
```

Em teoria, isso permite o uso de 65535 caracteres, no conjunto.

Para usar as funções do Windows da maneira mais performática possível você terá que lidar com essas strings estendidas... As funções da API que lidam com string têm um sufixo "W" no nome para indicar o tipo *Wide Char*, em posição ao *Ansi Char* (onde as funções têm um sufixo "A"). Por exemplo: A função *CreateWindow* tem as variações *CreateWindowW* e *CreateWindowA*.

No header *windows.h* existem macros que assumem, por default, os nomes de funções com sufixo "A" e, assim, você pode usar caracteres com 8 bits de tamanho nas strings. Mas essas funções converterão a string para UNICODE, internamente.

Outra coisa importante é que no mesmo header temos alguns tipos pré-definidos como LPWSTR e LPSTR. Eles são definidos como:

```
typedef wchar_t *LPWSTR; /* "Long Pointer to Wide-Char String" */
typedef char *LPSTR;     /* "Long Pointer to [Ansi] String" */
```

Digo isso porque, provavelmente, você não verá o tipo padrão *wchar\_t* sendo usado em códigos do Windows... Mas é a mesma coisa que LPWSTR. É ainda mais provável que você encontre um tipo chamao LPTSTR onde o "T" significa que o tipo será escolhido com base em símbolos definidos no ato da compilação (ou em alguma opção do compilador). Neste caso as strings literais poderão ser codificadas com o macro \_T:

```
LPTSTR str = _T("minha string");
```

Outros tipos que você pode encontrar são LPCSTR e LPCWSTR (e, acredito, o LPCTSTR), onde "C" signfica "const". Esses tipos são definidos como:

```
typedef char * const LPCSTR;
typedef wchar_t * const LPCWSTR;
```

Essas definições têm a pretensão de manter os códigos fonte, escritos em C, compatíveis entre as diversas versões do Windows. O termo "long" (o "L" no tipo) tem motivo histórico. Ele vem da época do Windows 3.1, onde ponteiros "near" e "far" eram comuns. Todo ponteiro, no Windows é "long".

## Vale a pena desenvolver aplicações inteiras em assembly para Windows?

Existem duas maneiras de usarmos funções definidas em DLLs, inclusive as definidas na Win64 API. A primeira é fazer uma importação "estática", deixando o Windows carregar a DLL para nós. A outra é carregá-la dinamicamente, através das funções *LoadLibraryW*, *GetProcAddressW* e *FreeLibraryW*.

No primeiro caso, o *MinGW64* disponibiliza um utilitário chamado *dlltool*, que cria uma biblioteca estática (que contém um arquivo objeto), contendo os símbolos para as funções definidas na DLL. Uma função como *MessageBoxW*, por exemplo, é associada a um símbolo nomeado *\_imp\_MessageBoxW*, que é um ponteiro para a função. Isso é fácil de observar, o pequeno código abaixo mostra:

```
/* msgbox.c */
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow)
{
    MessageBoxW(NULL, L"Minha String", L"Aviso", MB_OK | MB_ICONEXCLAMATION);
    return 0;
}
-----%----- corte aqui -----%-----
$ x86_64-w64-mingw32-gcc -O3 -S -masm=intel msgbox.c
```

```

-----%<---- corte aqui ---->%<-----
; Código equivalente, em asm:
bits 64
section .rodata

MinhaString  dw __utf16le__("Minha String"), 0
Aviso        dw __utf16le__("Aviso"), 0

section .text

extern __imp_MessageBoxW

global WinMain
align 16
_start:
    xor ecx,ecx
    lea edx,[MinhaString]
    lea r8,[Aviso]
    mov r9d,0x40      ; MB_OK | MB_ICONINFORMATION
    call [__imp_MessageBoxW]    ; Essas chamadas são sempre indiretas!
    xor eax, eax
    ret

```

A chamada para *MessageBox* é feita sempre de maneira indireta. O programa tem que saber o endereço da rotina, que **não** está localizada no seu programa, mas em *USER32.DLL* (mesmo no modo x86-64 o arquivo tem esse nome!).

A diferença do código em assembly e o código em C é que o GCC entende, graças ao header *windows.h*, que o ponto de entrada do programa é *WinMain()*. No caso do código em assembly o ponto de entrada, quando linkado usando o GNU Linker, continua sendo *\_start*. Fora esse fato, todo o resto segue exatamente o mesmo método usado por programas em C: É necessário linkar o seu objeto contra a biblioteca importada!

Já que não há diferenças e *syscalls* são inviáveis, por causa da falta de padronização, não vale lá muito a pena codificar programas inteiros em assembly para Windows.

Outra coisa: Num programa em C a *pseudo* função *WinMain* aceita 4 parâmetros (onde o segundo existe por motivos históricos e não é usado na Win64 API). O compilador insere código para inicializar esses parâmetros através de chamadas discretas a funções da API. Isso quer dizer que o compilador C prepara o terreno para você, coisa que não acontece em assembly.

## **Importando DLLs no MinGW-w64**

O MinGW64 tem um utilitário importante para a importação de funções contidas em DLLs. Trata-se do *dlltool*. Ele cria, entre outras coisas, uma biblioteca estática contendo as referências para a DLL desejada:

```
$ x86_64-mingw32-dlltool -D mydll.dll -z mydll.dll.def -l mydll.dll.a
```

Dlltool cria dois arquivos: Um com extensão .DEF, contendo a listagem dos símbolos exportados pela DLL e uma biblioteca estática (extensão .a, no padrão do GNU). O arquivo .DEF é textual e possui mais ou menos o formato:

```

LIBRARY nome-da-dll
EXPORTS
    símbolo1
    símbolo2
    ...

```

Ele é útil para que você saiba quais são os símbolos contidos na DLL desejada, mas não é necessário para linkarmos a biblioteca estática ao nosso projeto.

No exemplo do uso de *MessageBoxW*, poderíamos importar toda a *USER32.DLL*, criando o arquivo *user32.dll.a* (ou *user32.a*, você escolhe!) e usar a opção *-l* do linker para “linka-lo” ao

nosso programa.

Note que, no exemplo em C anterior, o símbolo `_imp_MessageBoxW` também é criado para permitir uma chamada indireta seja “mais direta”. Por “mais direta” quero dizer que normalmente a biblioteca estática cria funções “wraper” que tratam de fazer a chamada correta. O símbolo `_imp_MessageBoxW` provavelmente foi criado a partir de uma extensão do compilador, um atributo para funções chamado `_declspec(dllexport)`. No caso das funções “wraper” a chamada pode ser feita diretamente, mas o que você estará chamando é a função contida na biblioteca estática, não a função da API.

Usar bibliotecas estáticas para acessar funções de DLLs é um método conhecido como *early binding*, ou “juntar mais cedo”, numa tradução literal. Podemos “juntar mais tarde”, *late binding*, usando as funções da Win64 API `LoadLibraryW`, `GetProcAddressW` e `FreeLibrary`. Neste caso não precisamos nos preocupar obter uma biblioteca estática externa:

```
#include <windows.h>

int __attribute__((stdcall)) (*myfunc)(int);

/* Tenta carregar mylib.dll */
if ((hModule = LoadLibraryW(L"mylib.dll")) == NULL)
    return FALSE;

/* Tenta pegar o endereço de myfunc() */
if ((myfunc = GetProcAddressW(hModule, L"myfunc")) == NULL)
{
    FreeLibrary(hModule);
    return FALSE;
}

/* Finalmente, chama myfunc. */
x = myfunc(10);

FreeLibrary(hModule);
```



# Apêndice C: Built-ins do GCC

O GCC possui diversas funções já prontinhas para uso que permitem acesso a recursos do processador. Muitas delas são específicas para a arquitetura do processador alvo. Temos funções específicas para ARM, MIPS e outros. Aqui vou mostrar apenas algumas funções *built-in* que podem ser usadas para evitar a necessidade da criação de código em assembly.

Saiba que existem outras: **Todas** as instruções MMX, SSE, AVX, AVX2, FMA, BMI e um monte de outras extensões estão disponíveis via funções *built-in*, mas é conveniente que você continue usando as funções “intrínsecas”, disponibilizadas no header *x86intrin.h*.

Função	Descrição
<code>__builtin_cpu_is</code>	Retorna um booleano se a sua CPU é a informada na string:  <pre>if (!__builtin_cpu_is("intel")) {     puts("CPU não é Intel!");     exit(1); }</pre> Dentre as strings válidas, estão: “intel”, “amd”, “atom”, “core2”, “corei7”, “nehalem”, “sandybridge”, dentre algumas outras.
<code>__builtin_cpu_supports</code>	Retorna um booleano se sua CPU suporta a feature informada na string. O uso é similar à função <code>__builtin_cpu_is</code> , exceto que a string pode ser: “cmov”, “mmx”, “popcnt”, “sse”, “sse2”, “sse3”, “ssse3”, “sse4.1”, “sse4.2”, “avx” ou “avx2”.
<code>__builtin_expect</code>	Usada para dar uma dica ao <i>branch prediction</i> do processador. O valor 0 ou 1 no segundo parâmetro indica se o valor esperado pela expressão é verdadeiro ou falso:  <pre>/* Indica que, na maioria das vezes,    a expressão será verdadeira. */ if (__builtin_expect(x == 0, 1)) ...</pre> Deve ser usada com cautela, apenas nos casos mais críticos.
<code>__builtin_clear_cache</code>	Usada para liberar linhas do cache que contenham os endereços de início (primeiro parâmetro) até o fim (segundo parâmetro). A rotina provavelmente usa a instrução CLFLUSH para isso.  <pre>/* p1 e p2 são ponteiros onde p2 &gt; p1. */ __builtin_clear_cache(p1, p2);</pre>

	Deve ser evitada para não bagunçar muito o algoritmo de caching.
__builtin_prefetch	<p>Tenta fazer o <i>prefetching</i> no cache. A função aceita de 1 a 3 parâmetros. O primeiro é o endereço do dado desejado. O segundo é um booleano (0 ou 1), onde 1 significa “prefetching para escrita” e 0 (o default), óbviamente, “prefetching para leitura”.</p> <p>O terceiro parâmetro, se informado, indica a “localidade” ou “temporalidade”. O valor 0 indica “não temporalidade”, ou seja, o dado será usado muitas vezes no cache. Valores entre 1 e 3 indicam o nível de temporalidade. 3 (o default) significa alto nível de temporalidade e 1, baixo, onde um baixo nível significa que o dado poderá ser “esquecido” com mais facilidade.</p> <pre>/* x é colocado no cache para leitura    e temporalidade 3 */ __builtin_prefetch(&amp;x);  /* y é colocado no cache para escrita    e é não-temporal (durável). */ __builtin_prefetch(&amp;y, 1, 0);</pre> <p>De novo, evitar usar para não bagunçar o algoritmo de caching do processador.</p>
__builtin_bswap64	<p>Torna uma representação <i>little-endian</i> em <i>big-endian</i> ou vice-versa, mas com 64 bits. Funções como ntohs e htons faz o mesmo, mas com 32 bits (e também existe __builtin_bswap32):</p> <pre>/* Onde x e y são do tipo long    (ou "long long"). */ y = __builtin</pre>

No caso das funções \_\_builtin\_cpu\_is e \_\_builtin\_cpu\_support é necessário passar uma string para testar um recurso. Embora esse seja um método fácil, as strings terão que ser alocadas no segmento de dados (.rodata ou .data). Isso é algo que eu não aprecio... Acho preferível usar CPUID para obter o mesmo efeito. Abaixo, temos funções equivalentes, em C, para uso do GCC no modo x86-64:

```

/* cpu.c */
#include <memory.h>
#include <cpuid.h>

__attribute__((noinline)) static const char *get_cpu_id(void)
{
    static char cpuAux[12];
    int *p, a, b, c, d;

    __cpuid(0, a, b, c, d);
    p = (int *)cpuAux;
    *p++ = b;
    *p++ = d;
    *p = c;

    return cpuAux;
}

int cpu_is_intel(void) { return memcmp("GenuineIntel", get_cpu_id(), 12) == 0; }
int cpu_is_amd(void) { return memcmp("AuthenticAMD", get_cpu_id(), 12) == 0; }

/* Uso:
 *   if (get_cpu_features() & bit_SSE2) ...
 */
int get_simd_features(void)
{
    int a, b, c, d, _tmp;

    __cpuid(1,a,b,c,d);

    /* Os bits não se sobrepõem, então é seguro fazer um OR com os valores mascarados. */
    c &= bit_SSE3 | bit_SSSE3 | bit_FMA | bit_SSE4_1 | bit_SSE4_2 | bit_AVX | bit_F16C;
    d &= bit_SSE | bit_SSE2 | bit_MMX;
    _tmp = c | d;

    __cpuid_count(7,0,a,b,c,d);

    /* Os bits ainda não se sobrepõem e continua seguro fazer um OR. */
    b &= bit_BMI | bit_AVX2 | bit_BMI2;

    return _tmp | b;
}
-----%<---- corte aqui -----%>-----
/* test.c */
#include <cpuid.h>
#include <stdio.h>

/* definidas em cpu.c */
extern int cpu_is_intel(void);
extern int get_simd_features(void);

void main(void)
{
    printf("CPU ");
    if (!cpu_is_intel())
        printf(" não");
    printf(" é Intel.\n"
           "CPU ");
    if (!(get_simd_features() & bit_SSE2))
        printf(" não");
    puts(" suporta SSE2.");
}
-----%<---- corte aqui -----%>-----
# Makefile
test: test.o cpu.o
    gcc -O3 -o $@ $^

%.o: %.c
    gcc -O3 -march=native -c -o $@ $<

```



# Apêndice D: Módulos do Kernel

Na maioria das vezes estamos interessados em desenvolver programas que executam no *userspace*. Mas, alguns recursos só estão disponíveis no *kernelspace*.

De tempos em tempos topo com pessoas que querem desenvolver pequenos programas que interagem com a porta serial, ou outro dispositivo, de alguma maneira não padronizada. Querem, por exemplo, ter o poder de usar instruções IN e OUT, que normalmente não podem ser usadas no *userspace*. Para essas pessoas a melhor alternativa é desenvolverem um módulo para o kernel que funcionará como interface entre sua aplicação no *userspace* e o dispositivo.

## Anatomia de um módulo simples

Eis o código fonte de um módulo bem simples. As macros *module\_init* e *module\_exit* registram as funções de inicialização e término do módulo. Essas funções devem ser marcadas com os modificadores *\_init* e *\_exit*, respectivamente. As outras macros (*MOD\_LICENSE*, *MOD\_AUTHOR*, *MOD\_DESCRIPTION*) são usadas apenas para documentação:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init init_hello(void)
{
    printk(KERN_INFO "Olá, kernelspace!\n");
    return 0;
}

static void __exit exit_hello(void)
{
    printk(KERN_INFO "Adeus, kernelspace!\n");
}

module_init(init_hello);
module_exit(exit_hello);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Frederico Lamberti Pissarra");
MODULE_DESCRIPTION("Hello module");
```

Alguns detalhes importantes: Um módulo do kernel pode usar a *libc*, mas **não** deve fazê-lo... É preciso ter muito cuidado em usar funções reentrantes (marcadas com MT-Safe na documentação da *libc*) e tomar cuidado com condições que possam causar problemas com threads. Note, por exemplo, que a função *printk* foi usada ao invés de *printf*.