

Versão 2019

Execução de Programas:
Como funcionam carregadores,
ligadores e interpretadores

Bruno Müller Junior

Bruno Müller Junior

©2015-2019 Bruno Müller Junior

Este texto está licenciado sob a Licença Attribution-NonCommercial-ShareAlike 3.0 Unported da Creative Commons (CC). Em resumo, você deve creditar a obra da forma especificada pelo autor ou licenciante (mas não de maneira que sugira que estes concedem qualquer aval a você ou ao seu uso da obra). Você não pode usar esta obra para fins comerciais. Se você alterar, transformar ou criar com base nesta obra, você poderá distribuir a obra resultante apenas sob a mesma licença, ou sob uma licença similar à presente. Para ver uma cópia desta licença, visite <http://creativecommons.org/licenses/by-nc-sa/3.0/>.

Este texto foi produzido usando exclusivamente *software* livre: Sistema operacional Linux distribuição Mint, editor de texto Emacs, compilador de texto \LaTeX , gerenciador de referências \BibTeX , controlador de versão git, entre outros.

Lista de Figuras

Figura 1	– Espaço Virtual de um Processo	22
Figura 2	– Espaço Virtual de um Processo: Seções <i>Text</i> e <i>Data</i>	23
Figura 3	– Esquema de funcionamento das repetições <i>While</i> (esquerda) e <i>Repeat</i> (direita)	28
Figura 4	– Espaço Virtual de um Processo: Seção <i>Stack</i>	37
Figura 5	– Registro de ativação: modelo esquemático	39
Figura 6	– Registro de ativação: Informações de Contexto	41
Figura 7	– Registro de ativação: Variáveis Locais	44
Figura 8	– Montagem do registro de ativação com variáveis locais	46
Figura 9	– Desmontagem do registro de ativação com variáveis locais	46
Figura 10	– Registro de ativação: Parâmetros	47
Figura 11	– Passagem de parâmetros por referência	52
Figura 12	– Registros de ativação da execução do programa do Algoritmo 3.10	54
Figura 13	– Apontadores <i>argc</i> , <i>argv</i> e <i>arge</i>	63
Figura 14	– Espaço Virtual de um Processo: A região reservada ao sistema	69
Figura 15	– Código intruso no vetor de interrupção.	77
Figura 16	– Espaço Virtual de um Processo: Seção <i>BSS</i>	83
Figura 17	– Espaço Virtual de um Processo: <i>Heap</i>	89
Figura 18	– Alocador de Bartlett	93
Figura 19	– Conversão de Arquivos	101
Figura 20	– Organização de um arquivo no formato ELF Na esquerda, a organização para arquivos objeto e na direita, a organização para arquivos executáveis (adaptado de [25])	108
Figura 21	– Mapa de memória com um processo no formato COM em execução	124
Figura 22	– Mapeamento de um arquivo para execução em memória física	125
Figura 23	– Relocação de um arquivo no formato .EXE	128
Figura 24	– Organização de um arquivo no formato ELF Na esquerda, a organização para arquivos objeto e na direita, a organização para arquivos executáveis (adaptado de [25])	129
Figura 25	– Concatenação de arquivos objeto com uma única seção (esquerda) para criar um arquivo executável não relocado (centro) e relocado (direita).	134
Figura 26	– Concatenação de arquivos objeto com dois “pedaços” cada	135
Figura 27	– Espaço Virtual de um Processo: Segmento Dinâmico (ou compartilhado)	143
Figura 28	– Biblioteca compartilhada mapeada em dois processos (espaços virtuais de endereçamento diferentes).145	
Figura 29	– Execução do algoritmo 10.1	148
Figura 30	– Efeito da primeira chamada do procedimento “b()”, localizada em uma biblioteca compartilhada.	157
Figura 31	– Acesso às variáveis globais.	158
Figura 32	– Biblioteca compartilhada mapeada em dois processos.	164
Figura 33	– Acesso exclusivo às variáveis do processo usando paginação.	165
Figura 34	– Arquitetura de computadores baseados em fluxo de controle (arquitetura Von Neumann)	177
Figura 35	– Sintaxe da codificação de uma instrução (fonte:[46])	182
Figura 36	– Espaço Virtual de um Processo	191
Figura 37	– Arquitetura de computadores baseados em fluxo de controle (arquitetura Von Neumann)	192
Figura 38	– Todos os segmentos não cabem na memória física.	198
Figura 39	– Modelo de Memória Virtual	199
Figura 40	– MMU convertendo endereço virtual $(7120)_{16}$ para endereço físico $(2120)_{16}$	202
Figura 41	– Localização da MMU	204
Figura 42	– Mapeamento de dois processos na memória virtual.	204

Lista de Tabelas

Tabela 1	– Mapeamento do Programa 2.5 na memória (adaptado da saída do programa <code>objdump</code>)	29
Tabela 2	– Execução passo a passo do Programa 2.5 (adaptado da saída do programa <code>gdb</code>). As chaves à esquerda indicam execução no laço contido entre os rótulos <code>loop</code> e <code>fim_loop</code>	29
Tabela 3	– Funcionalidade das instruções <code>pushq</code> e <code>popq</code>	41
Tabela 4	– Funcionalidade das instruções <code>call</code> e <code>ret</code>	43
Tabela 5	– Tabela de mapeamento das variáveis do programa em C do Algoritmo 3.3 para as variáveis do programa <i>assembly</i> do Algoritmo 3.4.	45
Tabela 6	– Comparação das Figuras 8 e 9	47
Tabela 7	– Tradução de variáveis passadas por referência - Parte 1	50
Tabela 8	– Endereços das variáveis do programa do Algoritmo 3.8	50
Tabela 9	– Tradução de variáveis passadas por referência - Parte 2. Aqui, "a" é uma variável global e "x" é uma variável local	54
Tabela 10	– Convenção de associação de parâmetros e registradores	55
Tabela 11	– Uso de registradores de propósito geral (adaptado de [51])	58
Tabela 12	– Instruções para acessar os parâmetros em <code>main</code>	64
Tabela 13	– Convenção de uso dos registradores no MIPS (adaptada de [23])	66
Tabela 14	– Exemplos de chamadas de sistema (<code>syscalls</code>).	79
Tabela 15	– Associação de variáveis e seus locais de armazenamento no programa apresentado nos Algoritmos 5.2 e 5.3	86
Tabela 16	– Arquivos objeto e seus tamanhos	133
Tabela 17	– Instruções de desvio absoluto (linha 1) e relativo (linha 2) no AMD64	138
Tabela 18	– Funcionalidade das instruções de desvio absoluto e desvio indireto	148
Tabela 19	– Memória virtual do programa 10.7 em tempo de execução	168
Tabela 20	– Permissões das páginas da biblioteca <code>libMySharedLib.so</code> em tempo de execução	168
Tabela 21	– Relacionamento entre os números e nomes dos registradores	178
Tabela 22	– Números internos associados aos registradores AMD64	185
Tabela 23	– Tabela de mapeamento de endereços virtuais para endereços físicos: situação inicial	200
Tabela 24	– Tabela de mapeamento de endereços virtuais para endereços físicos: Após acesso à página virtual 7	200
Tabela 25	– Tabela de mapeamento de endereços virtuais para endereços físicos: Após acesso às páginas virtuais 01, 02, 04, 09, e 12	201
Tabela 26	– Relação entre endereços e as páginas	203

Lista de Algoritmos

2.1	Arquivo esqueletoC.c	24
2.2	Arquivo esqueletoS.s	24
2.3	Arquivo somaC.c	25
2.4	Arquivo somasS.s	26
2.5	Arquivo rotDesvioS.s	28
2.6	Comando while (Linguagem de alto nível)	30
2.7	Tradução do comando while da Figura 2.6 para assembly	31
2.8	Programa whileC.c	31
2.9	Tradução do programa whileC.c (Algoritmo 2.8) para assembly	31
2.10	Tradução do programa programa whileC.c (Algoritmo 2.8) sem acessos à memória	32
2.11	Comando If-Then-Else (Linguagem de alto nível)	33
2.12	Tradução do comando if-then-else do Algoritmo 2.11 para assembly	33
2.13	Exemplo para tradução	33
2.14	Arquivo maiorC.c	34
2.15	Arquivo maiorS.s	35
3.1	Programa sem parâmetros e sem variáveis locais	42
3.2	Tradução do programa do Algoritmo 3.1	42
3.3	Programa sem parâmetros e com variáveis locais	44
3.4	Tradução do programa do Algoritmo 3.3	45
3.5	Programa com parâmetros e com variáveis locais	48
3.6	Tradução do programa do Algoritmo 3.5	49
3.7	Programa com parâmetros passados por referência e com variáveis locais	50
3.8	Tradução do Algoritmo 3.7	51
3.9	Programa Recursivo	52
3.10	Tradução do Algoritmo 3.9	53
3.11	Reimpressão do programa recursivo do Algoritmo 3.9	56
3.12	Programa com um printf dentro de um for	56
3.13	Abordagem de salvar registradores pelo <i>caller</i>	57
3.14	Abordagem de salvar registradores pelo <i>callee</i>	58
3.15	Uso de printf e scanf	59
3.16	Tradução do Algoritmo 3.15	59
3.17	Uso de instruções <i>assembly</i> em C	60
3.18	Programa que imprime os argumentos e variáveis de ambiente	61
3.19	Programa que imprime argv[0]	64
3.20	O perigoso gets	65
3.21	Modificador static em variáveis locais	68
4.1	Carregamento dos endereços dos <i>drivers</i> no vetor	74
4.2	Direcionamento para o driver correto	75
4.3	Shell minimalista de [43]	80
5.1	Programa copiaC.c	84
5.2	Programa copiaS.s: primeira parte da tradução do algoritmo 5.1	87
5.3	Segunda parte da tradução do programa copiaS.s	88
6.1	Funcionamento da chamada <i>brk</i>	90
6.2	Exemplo de uso da API genérica de alocação de memória na <i>heap</i>	91
6.3	Programa que imprime os endereços de alocações sucessivas	92
6.4	Programa com <i>memory leak</i>	95
6.5	Overflow na <i>heap</i>	96
6.6	Fragmentação	96
6.7	Algo estranho	97

7.1	Arquivo esqueletoC.c	102
8.1	Estrutura do cabeçalho de um arquivo elf (em /usr/include/elf.h)	109
8.2	arquivo "a.c"	110
8.3	arquivo "b.c"	110
8.4	arquivo "c.c"	110
8.5	arquivo "d.c"	111
8.6	arquivo "main.c"	111
8.7	Estrutura Elf64_Shdr: cabeçalho das seções (em /usr/include/elf.h)	112
8.8	Estrutura Elf64_Phdr: cabeçalho das segmentos (de /usr/include/elf.h))	113
8.9	Estrutura ar_hdr: cabeçalho de um arquivo do tipo <i>archive</i> (de /usr/include/ar.h)	117
8.10	Arquivo a.c	119
8.11	Arquivo b.c	119
8.12	Arquivo main.c	120
9.1	Formato de um cabeçalho do formato .EXE[25]	127
9.2	Estrutura Elf64_Phdr: cabeçalho das segmentos (de /usr/include/elf.h))	130
9.3	Arquivo 01.c	133
9.4	Arquivo 02.c	133
9.5	Arquivo 03.c	133
9.6	Arquivo 04.c	134
9.7	Arquivo func_dep.c	141
9.8	Arquivo bar_dep.c	142
9.9	Arquivo simplemain.c	142
10.1	Programa que acessa as funções a partir de um veror de ponteiros	149
10.2	Arquivo a.c	152
10.3	Arquivo b.c	152
10.4	Arquivo main.c	153
10.5	Procedimento genérico da seção .plt	155
10.6	Procedimento zero@plt da seção .plt	156
10.7	Arquivo main.c	167
11.1	Programa bash com erro no else	173
B.1	Ação equivalente à da instrução <code>movq A(%rdi, 8), %rbx</code>	186
C.1	Reimpressão do arquivo esqueletoS.s	189
	Pgmas/sequenciaS.s	192

Prefácio

Em 1983 eu escrevi meus primeiros programas de computador. Na época, os programas eram digitados em cartão (uma linha por cartão) e executados no computador DEC-10 da Universidade Federal do Paraná, onde fiz o curso de Tecnólogo em Processamento de Dados. O curso era voltado ao desenvolvimento de *software*, com pouca ênfase no *hardware*.

Para minha (grata) surpresa, eu era bom nisto. Mesmo assim, não entendia como aqueles programas perfurados eram executados na máquina que só vi “funcionando” uma única vez¹.

Um primo sugeriu que eu comprasse um computador, mais especificamente um TK-82 com fabulosos 2 Kbytes de memória nativos e extensões de 16 Kbytes e 48 Kbytes. Ainda lembro dele dizendo “Você tem ideia do que você pode fazer com 2K?”. Fiquei quieto, mas a resposta era um rotundo não.

Adquiri um TK-85 com 16 Kbytes de memória nativos (uau!), e comecei a saciar minha curiosidade. O TK-85 tinha um teclado próprio, uma saída para TV e uma entrada e saída para fita cassete.

Os programas eram digitados (pacientemente) no teclado e salvos na fita cassete, o que permitia restaurar o que já tinha sido feito.

Uma vez minha mãe resolveu ouvir uma destas fitas (talvez preocupada com minha sanidade mental - afinal ela não conhecia ninguém que ficasse horas e horas na frente da TV sem assistir filmes). Ouviu um som assustador (algo semelhante àquele som de discagem de internet) e depois disto ela começou a me convidar a ir à igreja com alguma insistência.

A programação que eu fazia era na linguagem Basic, mas era incomum encontrar fitas com programas Basic à venda em lojas. O que eu encontrei - aos montes - eram fitas com joguinhos, todos desenvolvidos em linguagem de máquina. Nesta época eu fiz amizade com um “desenvolvedor” de jogos. Ele obtinha (comprava, importava) fitas de jogos, carregava na memória do TK-85, trocava o nome do desenvolvedor (para uma empresa inexistente), salvava em outras fitas e as vendia de loja em loja. Como era 1984, não me ocorreu perguntar se ele tinha um tapa-olho uma vez que o termo “pirata” só ficou conhecido bem mais tarde.

Ele tentou me ensinar a programar em linguagem de máquina do TK-85 (processador Z80). Era mais ou menos assim: “Deixe pressionadas estas duas teclas e digite aquela. Veja que apareceu 89 na tela, que é o código de operação para copiar ...”. Se você não entendeu, não se preocupe. Eu também não tinha entendido nada.

Porém, eu queria entender. Era algo semelhante à sensação de ver uma pauta musical, achar bonito e querer saber o que significa. Só que não foi desta vez.

Em 1986, fui trabalhar numa empresa que tinha computadores pessoais (PC) com MS-DOS. Um amigo me mostrou, empolgado, um bloco grosso de papel onde estava impresso só linguagem de montagem. As linhas continham algo como

```
89 E5 . . . . mov bp,sp
```

Ele estava empolgado e me disse “Veja: a instrução `mov bp,sp` é codificada em binário com os códigos hexadecimais 89 E5”. Do baixo de minha ignorância perguntei o que era aquela impressão, ao que ele me respondeu, quase aos pulos “É a impressão da BIOS”. Lembrei do meu primo e sorri dizendo “Ah, claro”.

Minha insatisfação por não entender aquilo era grande. Felizmente, este amigo me explicou muito e comecei a entender como um computador funciona e como os programas em linguagem de máquina “encaixam” no computador.

É claro que isto não me impedia de ficar admirado com o que ele era capaz de fazer. Porém não era o único com quem eu ficava admirado.

Certa vez outro amigo foi me buscar em minha sala (aos pulos de alegria, literalmente) e pediu para eu usar o computador dele. “Digite A:” ele me disse (isto acessava o disco flexível). O disco começou a fazer um barulho estranho, como se estivesse encharcado de água. Após algum tempo, apareceu a mensagem “Drive A está molhado”. Olhei para ele e ao olhar novamente para a tela, li: “Lavando” e o drive A fez barulho de máquina de lavar. Em seguida, “centrifugando” e o drive A fez barulho de centrifugar. Por fim, apareceu a mensagem “Drive A está pronto para o uso”.

Meio embaçado, ouvi ele dizer “É o meu vírus!” com um sorriso de orelha a orelha. Depois de algum tempo, juntei as peças: a BIOS está armazenada num trecho da memória física que não perde informações quando o computador

¹ Deduzo que estava funcionando porque muitas luzes coloridas estavam ligadas.

é desligado (memória ROM² ou EPROM³). Lá estão armazenadas as rotinas nativas de acesso aos periféricos. No caso, ele usou o trecho da BIOS que move o braço do disco para simular o barulho de disco molhado máquina de lavar (com temporização diferente em cada caso) e o trecho que girava o disco para simular o barulho de centrifugar.

Se você quer ter uma ideia do que estou falando, procure no seu buscador favorito por “Imperial march on floppy”, onde um princípio semelhante é usado para fazer com que disquetes “toquem” a marcha imperial de Star Wars. Existem variações em vários discos, impressoras matriciais, *scanner* entre outros. Tem até uma orquestra com 64 discos flexíveis, 8 discos rígidos e dois scanners que toca esta música. Se por um lado, fico admirado com o resultado, por outro fico pensando se este povo não tem mais nada para fazer.

Para tentar entender melhor aquele tema, era óbvio que eu precisava aprofundar meus conhecimentos em arquitetura de computadores e sistemas operacionais. Comprei um livro muito bom de S.O. chamado “Operating Systems Design and Implementation” de Andrew S. Tanenbaum [2]. Metade do livro era texto e a outra metade eram as 12.000 linhas do sistema operacional Minix, desenvolvido pelo autor para fins didáticos. O texto explicava conceitos como gerenciamento de processos e memória, paginação, acesso a dispositivos de entrada e saída, entre outros. Ao final de cada conceito, havia uma seção explicando como aquele conceito foi implementado no Minix, apontando para as linhas do código que continha a implementação.

Foi o primeiro livro de computação que li de capa a contracapa.

Do mesmo autor, li “Structured Computer Organization” [42] e “Computer Networks” [1] também de capa a contracapa. Nunca disse isto para minha mãe com medo que ela me forçasse a ir para a igreja com ela.

Minha curiosidade era muito grande, e tive felicidade de ser aceito em um mestrado com um tema que envolvia arquitetura de computadores e programação paralela. As coisas começavam a fazer sentido.

Em 1992, tornei-me professor do departamento de informática da UFPR, ministrei várias disciplinas naquela área que eu estudara com afinco, a área de sistemas, que inclui Construção de Compiladores, Sistemas Operacionais, Arquitetura de Computadores e Circuitos Lógicos. Isto permitiu que eu colocasse em prática tudo o que aprendi!

Uma característica comum a estas disciplinas é a dificuldade de expor todo o conteúdo no número de horas destinado a cada uma. Curiosamente, existem conceitos que são apresentados mais de uma vez com enfoques que atendem às particularidades de cada disciplina. Exemplos incluem chamadas ao sistema⁴, chamadas a procedimentos, uso da *heap*, uso de registradores e memória, execução dos programas entre outros. Além destes, aspectos como transformação de formatos de arquivos com montadores, ligadores, compiladores e carregadores, assim como os seus formatos, eram resumidos em função do pouco tempo disponível.

Como resultado, estes conceitos são fragmentados em partes que nem sempre são consistentes na visão de grande parte dos alunos. Além disto, não havia espaço para explicar os vários modelos de execução, carregadores, ligadores e montadores.

Para ensinar estes conceitos de uma forma mais consistente, o departamento de informática da UFPR criou a disciplina “Software Básico”, e este livro é a compilação das minhas notas de aula.

A disciplina aborda a camada de *software* mais próxima ao *hardware*, e explica alguns dos conceitos que são implementados parcialmente no *hardware* e como podem ser usados em *software*.

Para compreender como funciona esta camada, é necessário conhecer a linguagem de montagem⁵.

O objetivo do livro **NÃO É** ensinar os detalhes sadomasoquistas da programação em linguagem *assembly* (apesar de fazê-lo às vezes), mas sim usá-la para descrever como um programa é colocado em execução, descrevendo o formato de arquivos executáveis, arquivos objeto, bibliotecas além do funcionamento de programas interpretados, emuladores.

Desde que esta disciplina foi implantada, o tempo dispendido nas demais disciplinas de sistemas foi diminuído (o que não quer dizer que sobra tempo), e o nível de aprendizado demonstrado pelos alunos mostrou-se mais consistente.

² Read Only Memory

³ Erasable Programmable Read Only Memory

⁴ system calls

⁵ do inglês, linguagem *assembly*. Os dois termos são usados indistintamente ao longo do texto.

Sumário

Lista de Figuras	3
Lista de Tabelas	5
Lista de Algoritmos	7
Parte I Tradução e Execução de Programas	19
1 Tradução e Execução de Programas	21
2 A Seção Código e Dados	23
2.1 Esqueleto de programas em assembly	23
2.2 Expressões Aritméticas	25
2.3 Comandos Repetitivos	27
2.4 Tradução da construção While	30
2.5 Comandos Condicionais	32
2.6 Vetores	34
3 A Seção Pilha	37
3.1 Informações Armazenadas nas Chamadas de Procedimento	38
3.1.1 Escopo de Variáveis	38
3.1.2 Informações de Contexto	38
3.1.3 Registro de Ativação	39
3.2 Implementação do Modelo em Uma Arquitetura	40
3.2.1 Sem Parâmetros e sem Variáveis Locais	40
3.2.2 Sem Parâmetros e com Variáveis Locais	43
3.2.3 Com Parâmetros e com Variáveis Locais	46
3.2.4 Parâmetros passados por Referência e com Variáveis Locais	49
3.2.5 Chamadas Recursivas	51
3.3 Aspectos Práticos	55
3.3.1 Parâmetros passados em registradores	55
3.3.2 Como salvar variáveis em chamadas de procedimento	55
3.3.3 Uso de Bibliotecas	57
3.3.4 O procedimento main da linguagem C	60
3.3.5 Red Zone	63
3.3.6 Segurança	65
3.4 Registros de ativação no MIPS	66
4 Chamadas de Sistema	69
4.1 A analogia do parquinho	71
4.2 Acrescentando a CPU	72
4.3 Acrescentando periféricos e dispositivos de <i>hardware</i>	73
4.4 Interrupções	73
4.4.1 Interrupção de <i>Hardware</i>	74
4.4.1.1 Registrando o <i>driver</i>	74
4.4.1.2 Disparando o <i>driver</i> correto	75
4.4.1.3 Exemplo	75
4.4.2 Interrupção de <i>Software</i>	75
4.5 Os <i>drivers</i> nativos	76
4.6 Questões de segurança	77
4.7 As chamadas ao sistema no Linux	78
4.7.1 Conjunto de serviços	79
4.7.2 Sistema de Arquivos	80
4.7.3 Dispositivos Externos	81
5 A Seção BSS	83

6	A seção Heap	89
6.1	Gerenciamento da <i>heap</i>	90
6.1.1	Primeira tentativa	91
6.1.2	Segunda tentativa	91
6.1.3	Discussão	93
6.2	Funções de gerenciamento da <i>libc</i>	94
6.3	Problemas decorrentes	94
 Parte II Aplicativos Básicos		99
7	Aplicativos Básicos	101
8	Formatos de Programa	107
8.1	Arquivos em formato Objeto e Executável	107
8.1.1	Arquivos ELF em formato Objeto	109
8.1.2	Arquivos ELF em formato Executável	113
8.1.3	Bibliotecas	114
8.1.4	Bibliotecas Estáticas	115
8.1.5	Bibliotecas Compartilhadas	117
8.1.6	Bibliotecas Dinâmicas	118
8.1.7	Uma conversa sobre organização	121
9	O Carregador e o Ligador Estático	123
9.1	Carregadores Estáticos	123
9.1.1	Carregadores que copiam os programas em memória física sem relocação	124
9.1.2	Carregadores que copiam os programas em memória física com relocação	126
9.1.3	Carregadores que copiam os programas em memória virtual	129
9.2	Ligador para objetos estáticos	130
9.2.1	Ligadores Simples	131
9.2.2	Ligadores Com Relocação	131
9.2.3	Ligadores com seções	131
9.2.4	Exemplo Prático	132
9.2.5	Alocação de Espaço	132
9.2.5.1	Alocação de Espaço no ELF	134
9.2.6	Tabela de Símbolos	137
9.2.7	Relocação	138
9.2.8	Relocação no ELF	139
10	O Ligador Dinâmico	143
10.1	Vantagens e Problemas	144
10.2	Ferramentas do <i>Hardware</i>	146
10.2.1	Chamadas de procedimento e instruções de desvio	146
10.2.1.1	Desvio Relativo ao PC	146
10.2.1.2	Desvio Indireto	147
10.2.2	Instruções de acesso a dados	150
10.3	Implementação no ELF/Linux/AMD64	152
10.3.1	Exemplo de trabalho	152
10.3.2	Procedimentos	154
10.3.3	Variáveis Globais	157
10.3.3.1	Globais internas à biblioteca compartilhada	159
10.3.3.1.1	Trecho de código analisado	160
10.3.3.1.2	Análise do arquivo objeto	160
10.3.3.1.3	Análise em tempo de execução	160
10.3.3.2	Globais da biblioteca compartilhada visíveis externamente	161
10.3.3.2.1	Trecho de código analisado	162
10.3.3.2.2	Análise do arquivo objeto	163
10.3.3.2.3	Análise em tempo de execução	163
10.3.4	Acesso exclusivo às variáveis do processo	163
10.3.4.1	Visualizando a paginação no linux	166

11	Interpretadores	171
11.1	Emuladores	171
11.2	Interpretadores	172
11.2.1	Primeira Abordagem	172
11.2.2	Segunda Abordagem	173
11.2.3	Terceira Abordagem	173
11.2.4	Observações	174
Apêndices		175
APÊNDICE A Arquiteturas de Fluxo de Controle		177
A.1	Registradores	177
A.2	Memória	178
A.3	Conjunto de Instruções	178
A.3.1	Endereçamento Registrador	179
A.3.2	Endereçamento Imediato	179
A.3.3	Endereçamento Direto	180
A.3.4	Endereçamento Indireto	180
A.3.5	Endereçamento Indexado	180
A.4	Formato das Instruções	181
APÊNDICE B Formatos de Instrução		183
B.1	Modos de Endereçamento	183
B.2	Registrador	184
B.3	Imediato	185
B.4	Endereçamento Direto	185
B.5	Endereçamento Indireto	186
B.6	Endereçamento Indexado	186
B.7	Conclusão	187
APÊNDICE C Depurador		189
C.1	GDB	189
C.2	Sequenciamento da informação	192
C.3	Outros comandos de interesse	194
APÊNDICE D Memória Virtual		197
D.1	Overlay	197
D.2	Paginação	198
D.2.1	MMU	201
D.2.1.1	Mecanismos de Proteção	203
Referências		207

Organização do Livro

O livro está organizado em duas partes:

Primeira Parte: Linguagem assembly da família AMD64. A linguagem assembly é utilizada para exemplificar alguns dos conceitos de linguagens de programação, de sistemas operacionais e de arquitetura de computadores como, por exemplo, uso de registradores, memória, interrupções, chamadas ao sistema e de procedimento.

Segunda Parte: aplicativos de base (compiladores, ligadores, montadores e carregadores), os formatos intermediários dos arquivos que eles geram (arquivos objeto, arquivos executáveis, arquivos fonte).

Todos os programas apresentados neste livro foram desenvolvidos utilizando *software* livre desenvolvido pelo grupo GNU⁶ com um sistema operacional linux em máquinas família AMD64 (também conhecido como x86-64).

Os exemplos apresentados foram implementados e testados em um sistema operacional Linux 4.15.0-29, distribuição Linux Mint 19 (Tara). Para reproduzir os exemplos apresentados no livro, o leitor deve instalar os seguintes programas: `gcc` (compilador, montador, ligador), `as` (montador), `ld` (ligador), `gdb` (depurador), `readelf`, `objdump` (visualizadores de arquivos em formato ELF) e `ar` (gerador de biblioteca estáticas).

Eles estão disponíveis normalmente nos seguintes pacotes: `binutils`⁷, `gcc`⁸, `gdb`⁹.

Os códigos fonte dos programas apresentados no livro podem ser baixados de <https://www.inf.ufpr.br/bmuller/assets/docs/PgmaslivroSB.tar.bz2>.

Convenções

O texto adota algumas convenções:

- A arquitetura utilizada é denominada “AMD64”, que em outros lugares também é chamada “x64” e “x86-64”.
- Números hexadecimais são precedidos de `0x`, como por exemplo, `0xE9` ou com o subscrito 16, como em $(E9)_{16}$. Números binários usam o subscrito 2, como em $(E9)_{16} = (1110\ 1001)_2$.
- Para facilitar a leitura, números grandes são agrupados em 4 dígitos alinhados à direita, com zeros à esquerda como $(09AB\ CDEF)_{16}$ ou $(0001\ 1001)_2$.
- Comandos digitados na linha de comandos são destacados como abaixo:

```
> echo $HOME
/home/bmuller
>
```

⁶ <http://www.gnu.org/>

⁷ <http://www.gnu.org/software/binutils/>

⁸ <http://www.gnu.org/software/gcc/>

⁹ <http://www.gnu.org/software/gdb/>

Agradecimentos

Citar nomes é sempre injusto. Foram tantas as pessoas que contribuíram, que prefiro fazer agradecimentos gerais para poder incluir todos.

Agradeço aos vários alunos que contribuíram para que este livro fosse mais didático, e aos vários professores do departamento de informática que me incentivaram e que contribuíram para o aperfeiçoamento deste texto.

Porém, algumas pessoas eu tenho de citar. Primeiro ao meu colega e amigo Armando Delgado que leu este material mais de uma vez, analisando detalhadamente os exemplos e encontrando várias inconsistências que puderam ser corrigidas.

Agradeço em especial minha esposa e filhos, que quando me veem na frente do computador nunca tem certeza se estou trabalhando ou jogando. Sem sua paciência, este trabalho jamais teria sido possível.

Por fim, um agradecimento final aos meus dois filhos, Guilherme Galvão Müller e Gustavo Galvão Müller que elaboraram a capa sensacional deste livro. Se você não entende porque utilizei o termo “sensacional”, é porque você ainda não percebeu que a capa resume o livro.

Ah, e use o que QR Code para a versão mais recente do livro.

PARTE I

TRADUÇÃO E EXECUÇÃO DE PROGRAMAS

1. Tradução e Execução de Programas

Nesta parte do livro será descrito o que acontece ao longo do “tempo de vida” de um processo¹, ou seja, desde que ele inicia sua execução até ele terminar ou ser cancelado.

Quando um programa é colocado em execução, o sistema operacional é ativado para encontrar um espaço de memória onde este programa irá residir durante a execução².

A organização das informações de um processo na memória varia de um sistema operacional para outro. Os exemplos e conceitos apresentados neste livro foram desenvolvidos para o sistema operacional linux. Esta escolha se deve a vários fatores, dos quais destacamos:

1. o código fonte do sistema operacional pode ser obtido e estudado;
2. o formato dos arquivos objeto e executáveis, é livre e bem documentado (formato ELF³);
3. o modelo de execução é bem documentado[28].

A execução de programas será analisada em um nível próximo à CPU, e por esta razão, os programas usados para apresentar o modelo de execução foram desenvolvidos em linguagem assembly⁴. O assembly⁵ utilizado em todos os exemplos do livro foram codificados, montados e ligados em um sistema operacional Linux 4.15.0-29, distribuição Linux Mint 19 (Tara).

Como processador foi escolhida a família de processadores para 64 bits da AMD, também conhecidos como *AMD64*. Um motivo que levou a usar esta família de processadores é que ela é muito utilizada comercialmente, garantindo maior facilidade para reproduzir a execução dos programas descritos no livro.

O sistema operacional linux apresenta também algumas vantagens didáticas, uma vez que um programa em execução recebe uma grande quantidade de memória virtual. Mesmo podendo endereçar até 64 bits (um total de 2^{64} endereços), “somente” 2^{48} endereços são utilizados (0x0000 0000 0000 0000 até 0x0000 7fff ffff ffff).

Normalmente, o espaço de endereçamento é muito maior do que a quantidade de memória física. Para lidar com este problema, o sistema operacional linux utiliza um mecanismo para mapear os endereços da memória virtual para a memória física chamado paginação (veja Apêndice D).

A Figura 1 apresenta o espaço virtual de endereçamento de um programa em execução no modelo ELF do linux. O lado esquerdo apresenta todo o espaço endereçável, do endereço 0x0000 0000 0000 0000 até 0xffff ffff ffff ffff.

Como pode ser visto no lado esquerdo da Figura 1, este espaço é dividido em três segmentos:

Segmentos do Processo que vai do endereço 0x0000 0000 0000 0000 até o endereço 0x0000 7fff ffff ffff (128 TiB⁶). Esta área é para onde a imagem do processo é copiada, onde são alocadas suas variáveis, etc.

Segmentos Dinâmicos segmento de tamanho variável, onde objetos dinâmicos (procedimentos, variáveis, etc.) são alocados.

Reservado ao Sistema onde reside o Sistema operacional e outros objetos que o processo não pode acessar.

Este livro aborda todos os segmentos, a começar com o segmento de processo. O lado direito da figura mostra que este segmento é dividido em outras partes, ou seções, chamadas seção *text*, *data*, *bss*, *heap* e *stack*.

Cada uma destas seções será abordada em capítulos diferentes do livro. O Capítulo 2 descreve a seção de código (*text*), e de dados globais (*data*). O Capítulo 3 descreve como o programa em execução utiliza a pilha para implementar chamadas de procedimento, assim como alocar espaço para parâmetros e variáveis locais. O Capítulo 6 descreve como alocar espaço para variáveis dinâmicas (a *heap*). O Capítulo 4 descreve como o processo pode acessar recursos externos (como por exemplo, arquivos). Estes recursos são alocados através de pedidos explícitos ao sistema operacional (que reside no segmento reservado ao sistema apresentado na Figura 1 utilizando chamadas de sistema⁷).

¹ processo é o termo usado para referenciar um programa em execução.

² um programa não pode ser executado direto do disco (ou qualquer outro periférico).

³ Executable and Linkable Format

⁴ A tradução é linguagem de montagem, mas optamos pelo termo em inglês como padrão neste livro.

⁵ montador

⁶ 1 TiB (Tebibyte) = 2^{40} bytes

⁷ *System Calls*

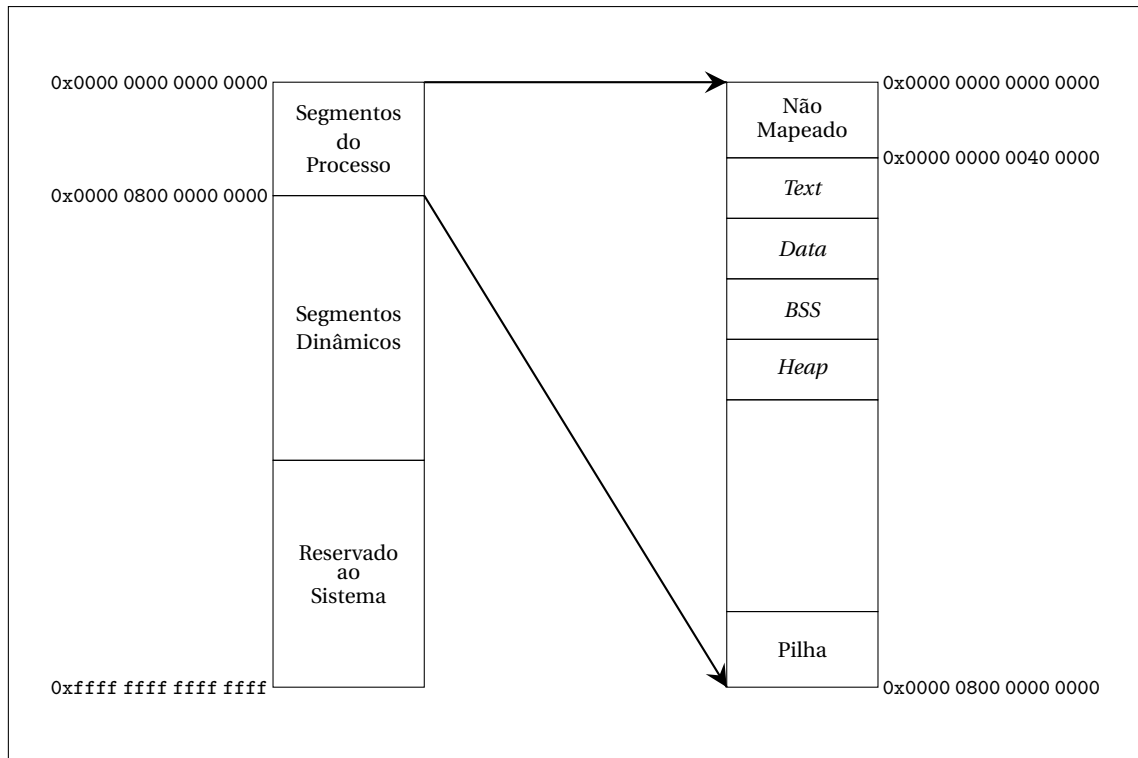


Figura 1 – Espaço Virtual de um Processo

O método didático utilizado neste livro baseia-se na tradução de pequenos programas apresentados na linguagem C para programas assembly *AMD64* **com funcionalidade equivalente** aos programas em C.

É importante destacar o termo “com funcionalidade equivalente”. Isto significa dizer que o programa assembly produzirá os mesmos resultados que o programa C geraria, porém não obrigatoriamente com os mesmos comandos gerados por um compilador, e nem com os mesmos endereços de variáveis.

Programas assembly são programas com muitas linhas de código, e quase sempre difíceis de serem compreendidos por seres humanos (ou pelo menos pela maioria deles). Este livro não tem a intenção de ensinar a linguagem assembly, e os exemplos são apresentados unicamente para descrever o que ocorre durante a execução de programas.

Por esta razão, não é objetivo do livro entrar em detalhes e muito menos de apresentar o conjunto completo de instruções contido na família *AMD64*⁸. Será descrito somente um subconjunto mínimo de instruções para os fins propostos pelo livro. Este subconjunto exclui várias instruções e formatos de instrução, o que faz com que vários dos programas sejam mais longos do que precisariam ser.

Este subconjunto não é apresentado de uma vez, mas sim de forma incremental. Cada exemplo inclui algumas instruções ao conjunto apresentado até aquele momento, seguindo o modelo utilizado por Kowaltowski em [22].

⁸ O autor deste livro entende que isso seria desumano. Aliás, isto pode dizer muito sobre os professores que adotam esta abordagem.

2. A Seção Código e Dados

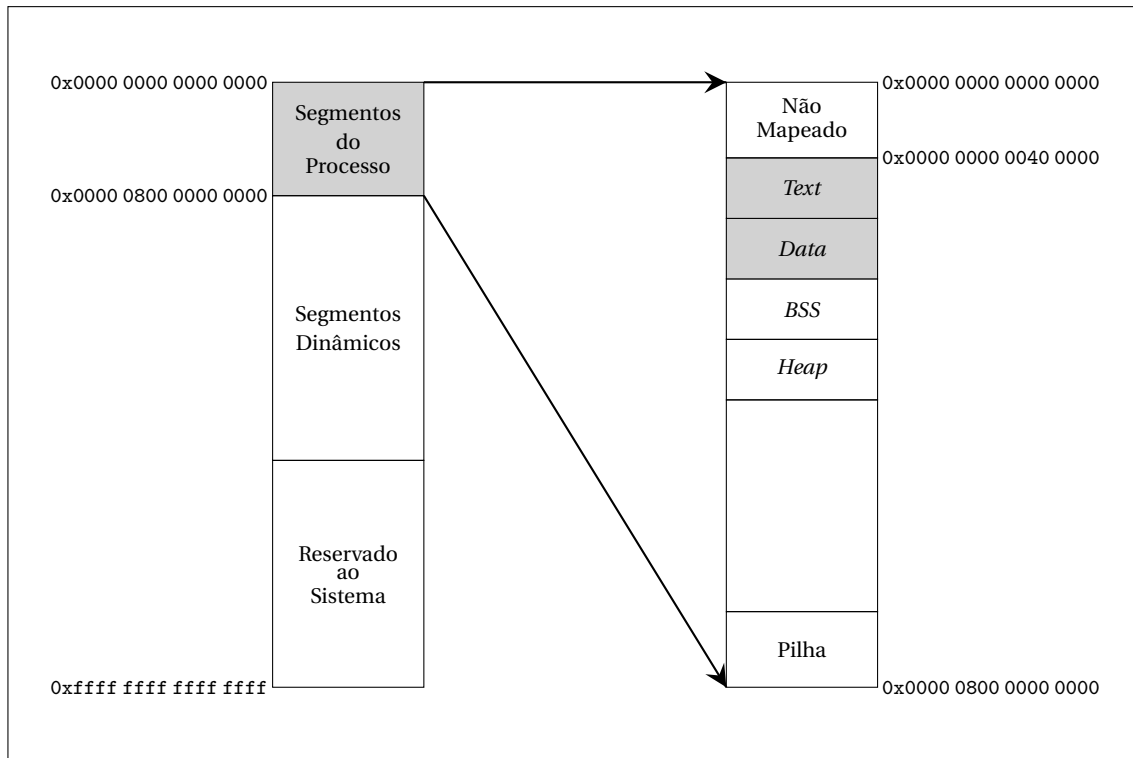


Figura 2 – Espaço Virtual de um Processo: Seções Text e Data

Este capítulo trata das seções `text` e `data`, que estão destacadas na Figura 2. A seção `text` contém as instruções do programa, enquanto que a seção `data` contém as variáveis globais.

A Figura 2 mostra que:

- A seção `text` inicia no endereço virtual 0x0000 0000 0040 0000, mas não tem um endereço de término indicado.
- a seção `data` não tem um endereço de início e de fim especificados (veja Exercícios 2.4 e 2.5).

O capítulo aborda estas duas seções apresentando programas em C e os programas equivalentes em assembly. Os programas são apresentados em nível de complexidade crescente. Cada seção acrescenta um conjunto de instruções assembly àquele conjunto já apresentado anteriormente.

O capítulo está organizado da seguinte forma: a Seção 2.1 apresenta um “esqueleto de programa” assembly. A Seção 2.2 descreve como traduzir expressões aritméticas utilizando variáveis globais. A Seção 2.3 apresenta o funcionamento de comandos de desvio de fluxo em linguagem assembly, enquanto que a Seção 2.4 mostra como traduzir comandos repetitivos do tipo `while`. A Seção 2.5 descreve como traduzir comandos condicionais do tipo `if-then` e `if-then-else`. Para finalizar, a Seção 2.6 apresenta um exemplo de acesso à memória combinando todas as construções apresentadas no capítulo.

2.1 Esqueleto de programas em assembly

Observe o programa escrito em linguagem C no Algoritmo 2.1. Ele não faz nada útil, só termina graciosamente. O programa tem poucos comandos e por isso mesmo é apropriado para apresentar o esqueleto de um programa assembly equivalente e posteriormente alguns aspectos da execução de programas.

```

1 int main (long int argc, char **argv)
2 {
3     return 13;
4 }

```

Algoritmo 2.1 – Arquivo esqueletoC.c

O único comando do programa está na linha 3: `return 13`. Este comando finaliza o programa e retorna o número 13 para quem o invocou (neste caso para o programa *shell*), como pode ser visto abaixo.

```

> gcc esqueletoC.c -o esqueletoC
> ./esqueletoC
> echo $?
13
>

```

Aqui, o Algoritmo 2.1 está contido no arquivo `esqueletoC.c`. A primeira linha (`gcc esqueletoC.c -o esqueletoC`) gera o arquivo executável `esqueletoC` a partir do arquivo `esqueletoC.c`. A segunda linha (`./esqueletoC`) executa o programa executável `esqueletoC`. Quando este programa finaliza, a *shell* captura o valor retornado pela linha 3 (13) e o armazena na variável de ambiente `$?`. A última linha `echo $?` imprime o conteúdo da variável de ambiente `$?`, ou seja, 13.

Agora vamos tratar do programa assembly. Existem várias formas de escrever um programa equivalente em assembly do AMD64. Um exemplo é o apresentado no Algoritmo 2.2.

```

1 .section .data
2 .section .text
3 .globl _start
4 _start:
5     movq $60, %rax          # %rax := 60
6     movq $13, %rdi         # %rdi := 13
7     syscall

```

Algoritmo 2.2 – Arquivo esqueletoS.s

Como é o primeiro programa assembly, algumas explicações são necessárias:

- As linhas que começam com ponto (`.section` e `.globl`) são diretivas que orientam o montador¹ (nome dado ao programa que lê assembly como entrada e gera código objeto como saída). A diretiva `.section .data` indica que ali inicia a lista de variáveis globais do programa. Como neste caso não há nenhuma, ela fica vazia e poderia até ser omitida. A diretiva `.section .text` indica que seguem os comandos assembly daquele programa. A diretiva `.section .globl` diz que os símbolos que seguem são símbolos que devem ser conhecidos externamente.
- `_start` é um rótulo. Programas assembly não tem estruturas de controle, como `while`, `repeat`, `if`, etc. O fluxo de execução só pode ser alterado com instruções explícitas de desvio cujo alvo é um endereço, no formato `desvie <endereço>`. Porém, para simplificar a vida do programador assembly, o endereço pode ser um símbolo, chamado rótulo. Voltaremos a este assunto mais adiante.

A tradução não é literal, uma vez que o `main` é uma função e deveria ter sido traduzida como tal (o que será visto no Capítulo 3). Porém, é uma tradução válida, uma vez que a funcionalidade do programa assembly equivale à funcionalidade do programa em C.

Observe que o parâmetro `argc` foi declarado como `long int`. Neste exemplo, poderia ser declarado como `int`, ou até omiti-lo junto com `argv`. Porém, optamos por declará-lo como `long int` porque isto diz ao compilador que esta variável ocupa oito bytes (64 bits), mesmo tamanho dos registradores do AMD64². Ao longo do livro, todas as variáveis inteiras serão declaradas como `long int` para simplificar o mapeamento com registradores e com os endereços de memória das variáveis.

¹ *assembler*

² uma variável declarada como `int` ocupa quatro bytes, 32 bits.

Programas em assembly são divididos em seções, e o programa apresentado contém duas seções. A seção `.data` contém as variáveis globais do programa (como este programa não usa variáveis globais, esta seção está vazia) enquanto que a seção `.text` contém os comandos a serem executados quando o programa for colocado em execução.

A linha 4 do Algoritmo 2.2 contém um rótulo, ou seja, um nome associado a um endereço. Na sintaxe do assembly que estamos usando (como em vários outros), o rótulo é sempre um conjunto de caracteres e letras terminadas por dois pontos.

O rótulo `_start` é especial e deve sempre estar presente em um programa assembly. Ele corresponde ao endereço da primeira instrução do programa que será executada, e deve ser declarada como global (linha 3). As demais instruções serão executadas na sequência.

Os comandos necessários para gerar o arquivo executável estão descritos a seguir:

```
> as esqueletoS.s -o esqueletoS.o
> ld esqueletoS.o -o esqueletoS
> ./esqueletoS
> echo $?
13
```

O programa `as`³ converte um programa escrito em assembly (`prog1.s`) num arquivo objeto (`prog1.o`). Como será visto na segunda parte deste livro, um arquivo objeto é uma versão incompleta de um arquivo executável. O programa `ld`⁴ combina arquivos objeto e gera um arquivo executável (no caso `prog1`). O comando `./prog1` invoca o sistema operacional que o coloca em execução.

Uma vez explicado como gerar um arquivo executável e como executá-lo, iremos analisar com mais detalhes o que ocorre durante a execução. A ideia é mostrar a execução instrução a instrução (assembly) e o efeito de cada instrução, ou seja, o que a instrução altera na memória e na CPU.

Para entender esta execução passo a passo, é necessário saber o que é a memória, o que é a CPU e que papel eles têm na execução. O Apêndice A apresenta uma descrição rápida da CPU do AMD-64 assim como seu relacionamento com a memória e dispositivos de entrada e saída.

Para ver o que ocorre durante a execução será utilizado um depurador. Este livro utiliza o “gdb”⁵, um depurador em modo texto. O Apêndice C apresenta uma rápida introdução ao “gdb” mas qualquer depurador que mostre a execução passo a passo (em assembly) e os valores dos registradores e memória ao longo da execução pode ser utilizado.

Recomenda-se fortemente a leitura do Apêndice C antes de continuar.

2.2 Expressões Aritméticas

Esta seção mostra como traduzir para assembly os programas escritos em linguagem C que contém somente expressões aritméticas que usam variáveis globais.

O Algoritmo 2.3 é um programa em linguagem C que utiliza duas variáveis globais (`a` e `b`). Estas variáveis são somadas e o resultado é colocado para o usuário (na variável de ambiente `?` do shell).

```
1 long int a, b;
2 int main ( long int argc, char **argv)
3 {
4     a=7;
5     b=6;
6     b = a+b;
7     return b;
8 }
```

Algoritmo 2.3 – Arquivo somaC.c

O Algoritmo 2.4 corresponde ao programa assembly equivalente ao programa C do Algoritmo 2.3.

O primeiro aspecto a ser levantado é como as variáveis globais “a” e “b” foram declaradas no programa assembly (linhas 2 e 3 do Algoritmo 2.4, ou seja, como rótulos (veja a definição de rótulo na página 25).

³ the portable GNU assembler

⁴ The GNU linker

⁵ Gnu Debugger <https://www.gnu.org/software/gdb/>

```

1  .section .data
2      A: .quad 0
3      B: .quad 0
4  .section .text
5  .globl _start
6  _start:
7      movq $6, A
8      movq $7, B
9      movq A, %rax
10     movq B, %rbx
11     addq %rax, %rbx           # %rbx := %rbx + %rax
12     movq $60, %rax
13     movq %rbx, %rdi
14     syscall

```

Algoritmo 2.4 – Arquivo somaS.s

Estes dois rótulos foram declarados na seção “.data”. Quando o programa for colocado em execução, serão reservados dois espaços de inteiro de 64 bits (para isto o .quad das linhas 2 e 3 do Algoritmo 2.4). O parâmetro que vem após o .quad indica o valor inicial da variável, que no caso é zero. A inclusão de um valor é necessária para que o rótulo A e o rótulo B sejam colocados em endereços diferentes.

Para deixar este ponto mais claro, vamos verificar como o programa será executado. Após montar e ligar o programa, obteremos o arquivo executável somaS. Vamos utilizar novamente o simulador gdb e analisar a segunda instrução (linha 8).

```

...
4000b0:      48 c7 04 25 e7 00 60      movq    $0x7,0x6000e7
4000b7:      00 07 00 00 00
4000bc:      ...

```

A primeira coluna indica endereços. A segunda, o conteúdo de cada endereço e a última coluna indica o mnemônico daquela instrução.

Assim, a instrução com mnemônico movq \$0x7,0x6000e7, quando representada em código de máquina, é composta por 12 bytes: 0x48c7 0425 e700 6000 0700 0000 que, em tempo de execução serão mapeados em memória a partir do endereço 0x0040 00b0. Se considerar que a memória é um vetor M[] isto pode ser visto como indicado abaixo:

```

...
M[0x004000b0] = 0x48
M[0x004000b1] = 0xc7
M[0x004000b2] = 0x04
M[0x004000b3] = 0x25
...

```

Observe que a instrução movq \$0x7,0x6000e7 começa no endereço 0040 00b0 e vai até o endereço 0040 00bb. No endereço seguinte (0040 00bc) começa a representação em código de máquina da próxima instrução.

Ao examinar o Algoritmo 2.4, constata-se que esta instrução em código de máquina corresponde à instrução da linha 8, ou seja, movq \$7, A.

Desta forma, é possível deduzir que, em tempo de execução, o endereço de “A” é 0x0060 00e7.

Como será visto adiante, quem define o endereço de “a” e de “b” é o ligador, e não o montador e nem o carregador. Para comprovar isso, observe o resultado abaixo, do conteúdo do arquivo objeto (gerado pelo montador).

```

> objdump somaS.o -S
...
0000000000000000 <_start>:
0:      48 c7 04 25 00 00 00      movq    $0x7,0x0
7:      00 07 00 00 00

```

```

c:      48 c7 04 25 00 00 00      movq    $0x6,0x0
13:      00 07 00 00 00
...

```

Compare este resultado com o conteúdo do arquivo executável (gerado pelo ligador):

```

> objdump -S somaS
...
00000000004000b0 <_start>:
4000b0:      48 c7 04 25 e7 00 60      movq    $0x7,0x6000e7
4000b7:      00 07 00 00 00
4000bc:      48 c7 04 25 ef 00 60      movq    $0x6,0x6000ef
4000c3:      00 06 00 00 00
...

```

Em especial, examine a instrução `movq $7, a` nos dois arquivos.

O termo “variável” em assembly é diferente do termo utilizado em linguagens de programação fortemente tipadas (como Pascal), onde cada variável é do tipo declarado desde o “nascimento” da variável até sua “morte” (ou seja, ao longo do seu tempo de vida). Nestas linguagens, uma variável do tipo inteiro não pode ser somada a uma variável real (apesar de ambas poderem ocupar o mesmo número de bits).

Já no caso de assembly, as variáveis não estão associados a nenhum tipo (.quad indica que deve ser alocado espaço necessário para um inteiro, ou seja, 64 bits). Isto não significa que “A” e “B” só possam ser usados em operações inteiras. Nem o montador nem o ligador fazem checagem de tipo - isto é responsabilidade do programador (e volta e meia isto ocasiona grandes dores de cabeça).

Outro ponto importante é que a instrução de soma utilizada, `addq`, que soma o conteúdo de dois registradores (neste caso `%rax` e `%rbx`) e coloca o resultado em `%rbx`. Esta operação soma dois inteiros de 64 bits com ou sem sinal. O caso de ocorrer `overflow`⁶ (em qualquer um dos dois casos) é indicado no registrador `RFLAGS`, que será abordado mais adiante.

Para outros tipos, formatos de somas, e outras instruções para operações aritméticas, veja [46].

2.3 Comandos Repetitivos

Comandos repetitivos são aqueles que permitem que um conjunto de instruções seja repetido até que uma determinada condição ocorra. Em linguagens de alto nível, normalmente são divididos em comandos do tipo “repeat” e comandos do tipo “while”. Um “repeat” indica que o conjunto de instruções pode ser repetido de uma a várias vezes, enquanto que um “while” indica que o conjunto de instruções pode ser repetido de zero a várias vezes. A forma de operação destas construções está apresentada esquematicamente na Figura 3.

Há também a construção do tipo “for”, que indica que um conjunto de instruções deve ser repetido um determinado número de vezes. Esta construção é semelhante ao comando `while`, porém acrescida de uma variável de controle e de:

1. um comando para iniciar a variável de controle antes do primeiro teste;
2. um comando para incrementar de um a variável de controle após a execução da sequência de comandos especificada;
3. uma expressão que verifica se o valor da variável de controle ultrapassou o limite.

Em linguagens assembly, estas construções não estão presentes, o nosso próximo passo é descrever como traduzir comandos de “alto nível” para comandos de “baixo nível”. Esta tradução é feita automaticamente pelos compiladores, mas nem sempre são exatamente como descritas aqui principalmente para melhorar o desempenho.

Como a Figura 3 mostra, existem locais onde o fluxo de execução é bifurcado, assim como locais onde o fluxo não segue para a próxima instrução.

As instruções vistas até aqui indicam (implicitamente) que a instrução seguinte é a próxima a ser executada (na memória). Para implementar estas mudanças de fluxo em assembly, são necessárias instruções especiais.

Estas instruções assembly permitem desviar o fluxo de execução utilizando rótulos como destino. Como exemplo, considere o Algoritmo 2.5, que incrementa o valor em `%rax` de um até ultrapassar o valor de `%rbx`. Este programa tem três

⁶ `overflow` ocorre quando o resultado da operação precisa de mais bits do que o disponibilizado pela máquina. Por exemplo, em máquinas de 4 bits o resultado da soma: $(1111)_2 + (0001)_2 = (10000)_2$ é zero $((15)_{10} + (1)_{10} = (0)_{10})$ pois o bits à esquerda é desprezado.

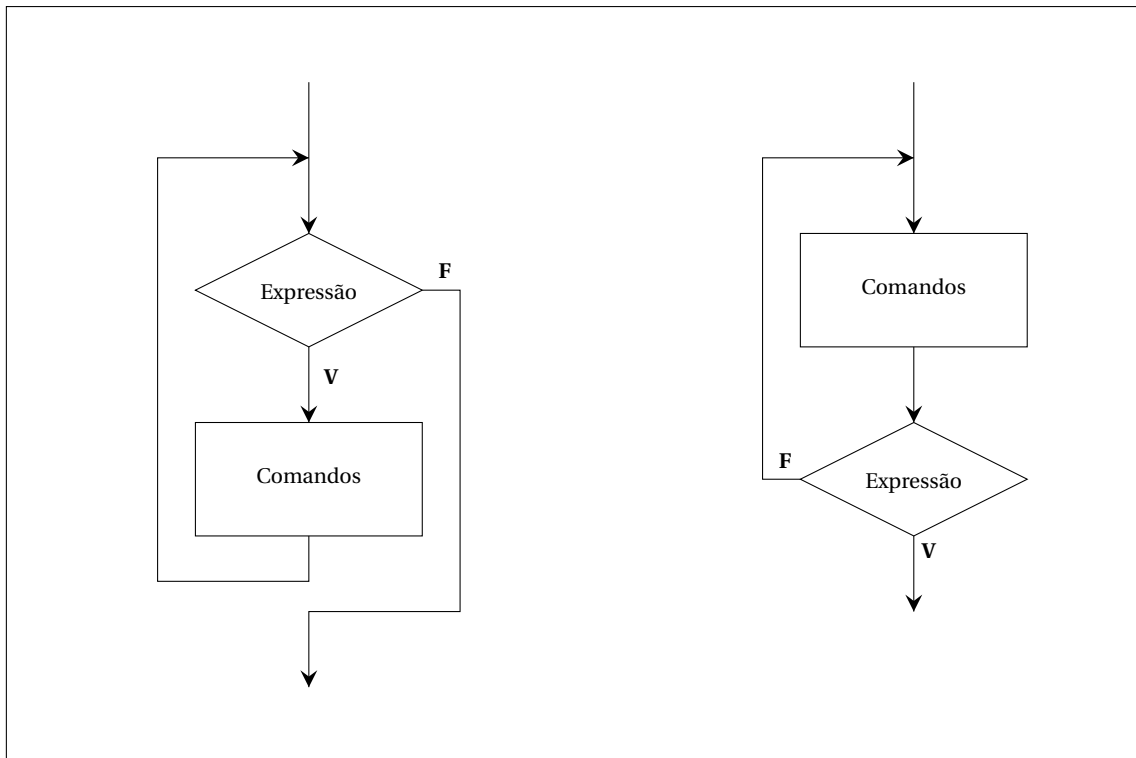


Figura 3 – Esquema de funcionamento das repetições While (esquerda) e Repeat (direita)

rótulos (`_start`, `loop`, `fim_loop`) e duas instruções de desvio (`jg` e `jmp`), que estão para desviar se maior (*jump if greater*) e desvio incondicional (*jump*) respectivamente.

O rótulo `_start`, como já vimos, indica o local onde a execução do programa deve iniciar. O rótulo `loop` indica o ponto de início do laço e o rótulo `fim_loop` indica o ponto de saída do laço, que será executada assim que a condição de término for atingida.

```

1  .section .text
2  .globl _start
3  _start:
4      movq $0, %rax
5      movq $10, %rbx
6  loop:
7      cmpq %rbx, %rax
8      jg fim_loop
9      add $1, %rax
10     jmp loop
11 fim_loop:
12     movq $60, %rax
13     movq %rax, %rdi
14     syscall

```

Algoritmo 2.5 – Arquivo `rotDesvioS.s`

Quando o Programa 2.5 é colocado em execução, cada instrução é armazenada em um endereço da memória virtual como apresentado na Tabela 1.

Observe que o programa executável já contém os endereços virtuais que serão ocupados quando o programa for carregado para execução. Verifique que os endereços da seção `.text` estão compatíveis com o que foi indicado na Figura 2.

Rótulo	Endereço	instrução (hexa)	instrução
_start	0x0040 0078	48 c7 c0 00 00 00 00	mov \$0x0,%rax
	0x0040 007f	48 c7 c3 0a 00 00 00	mov \$0xa,%rbx
loop	0x0040 0086	48 39 d8	cmp %rbx,%rax
	0x0040 0089	7f 06	jg 400091 <fim_loop>
	0x0040 008b	48 83 c0 01	add \$0x1,%rax
	0x0040 008f	eb f5	jmp 400086 <loop>
fim_loop	0x0040 0091	48 c7 c0 3c 00 00 00	mov \$0x3c,%rax
	0x0040 0098	48 89 df	mov %rsx,%rdi
	0x4000 009b	0f 05	syscall

Tabela 1 – Mapeamento do Programa 2.5 na memória (adaptado da saída do programa objdump)

Além das instruções de desvio, o programa apresenta um outro comando novo: `cmpq`. Esta instrução compara o SEGUNDO argumento com o primeiro (no caso, `%rax` com `%rbx`) colocando o resultado em um bit de um registrador especial (RFLAGS). Este registrador é afetado por vários tipos de instrução, e contém informações sobre a última instrução executada, como por exemplo, se ocorreu overflow após uma soma.

Os bits deste registrador podem ser testados individualmente, e no caso da operação `jump if greater`, verifica se o bit ZF (zero flag) é igual a zero e se SF=OF (SF=Sign Flag e OF=Overflow Flag). Para mais detalhes sobre o funcionamento do RFLAGS, veja [45]

Observe que se alguma outra instrução for colocada entre a comparação (`cmpq`) e o desvio, os bits do RFLAGS poderão afetados por esta nova instrução. Isto implica dizer que a instrução de desvio `jg` deve ser colocada imediatamente após a instrução `cmpq`.

Outras instruções de desvio que serão utilizadas ao longo deste livro são:

- `jge` (*jump if greater or equal*),
- `jle` (*jump if less*),
- `jle` (*jump if less or equal*),
- `je` (*jump if equal*),
- `jne` (*jump if not equal*).

Assim como a instrução `jg`, as instruções acima também funcionam analisando os bits de RFLAGS.

Como exercício, simule a execução do programa acima, prestando atenção a registrador `%rip`. Fiz isto em meu simulador, e obtive o resultado apresentado na Tabela 2.

	%rip	instrução (hexa)	instrução
1	0x40 0078	48 c7c0 0000 0000	movq \$0x0,%rax
2	0x40 007f	48 c7c3 0a00 0000	movq \$0xa,%rbx
3	0x40 0086	48 39d8	cmpq %rbx,%rax
4	0x40 0089	7f06	jg 0x400091
5	0x40 008b	4883 c001	addq \$0x1,%rax
6	0x40 008f	ebf5	jmp 0x400086
7	0x40 0086	48 39d8	cmpq %rbx, %rax
8	0x40 0089	7f06	jg 0x400091
9	0x40 008b	4883 c001	add \$0x1,%rax
10	0x40 008f	ebf5	jmp 0x400086
11	0x40 0086	48 39d8	cmpq %rbx, %rax
...	

Tabela 2 – Execução passo a passo do Programa 2.5 (adaptado da saída do programa gdb). As chaves à esquerda indicam execução no laço contido entre os rótulos `loop` e `fim_loop`.

Cada linha indica a execução de exatamente uma instrução. A primeira coluna da esquerda indica a ordem de execução das instruções. A coluna `%rip` indica o endereço da instrução executada, e a terceira coluna indica o conteúdo

daquele endereço (a instrução em hexadecimal). A última coluna mostra o código mnemônico da instrução hexadecimal indicada.

Compare a Tabela 2 com a Tabela 1. Analise em especial a sequência de execução e que o registrador `%rip` é quem indica qual a instrução a ser executada.

A primeira instrução executada estava no endereço `0x400078` (endereço do rótulo `_start`). A instrução contida naquele endereço é `movq 0x0,%rax`, cujo código de máquina (em hexadecimal), é `48 c7 c0 00 00 00 00`.

Como esta instrução ocupa sete bytes, a próxima instrução deverá estar localizada em `0x40 0078 + 0x7 = 0x0x40 007f`. Este é o valor de `%rip` ao longo da execução da instrução contida no endereço `0x400078` (o `%rip` sempre aponta para a próxima instrução nos computadores baseados no modelo Von Neumann, como o AMD64, conforme explicado no Apêndice A).

A Tabela 2 mostra duas iterações no laço do Programa 2.5. O primeiro laço está indicado nas linhas 3 até 6 e o segundo nas linhas 7 até 10, destacadas pelas chaves à esquerda. Compare as linhas 3 com 7, 4 com 8, 5 com 9, 6 com 10, 7 com 11. Elas indicam os mesmos valores de `%rip`, ou seja, a repetição da execução das instruções que localizam-se naqueles endereços de memória o que é de se esperar que ocorra em um laço.

Em quase toda a tabela, percebe-se que o endereço da próxima instrução é obtido ao somar o endereço da instrução atual com o tamanho desta instrução, algo como

$$\%rip \leftarrow \%rip + (\text{tamanho da instrução})$$

Esta lógica de funcionamento é quebrada quando ocorre um desvio. Observe a coluna `%rip` após a instrução `jmp loop` (linhas 6 e 10). A instrução seguinte está no endereço `0x40 0086`, que não coincidentemente é o mesmo endereço do rótulo `loop`, e indicado explicitamente nas linhas 6 e 10 (`jmp 0x40 0086`) onde `loop` foi substituído pelo endereço onde está o rótulo (`0x40 0086`).

Nestes casos, temos algo como

$$\%rip \leftarrow \text{endereço do rótulo}$$

Um rótulo sempre indica um endereço na memória em tempo de execução. Neste exemplo, rótulos são utilizados para indicar endereços que são alvos de desvio do fluxo de execução.

2.4 Tradução da construção While

A tradução de uma construção `while` segue o fluxo de execução apresentado na Figura 3, onde os desvios no fluxo são substituídos por comandos de desvio em assembly e os “pontos de entrada” dos desvios são substituídos por rótulos.

Como exemplo, considere os Algoritmos 2.6 e 2.7.

```

1 int main ( long int argc, char **argv)
2 {
3   ...
4   while ( E ) {
5       C1; C2; ... Cn;
6   }
7   ...
8 }
```

Algoritmo 2.6 – Comando `while` (Linguagem de alto nível)

As instruções nas linhas 5, 7, 9 e 10 do Algoritmo 2.7 correspondem às instruções que dão a funcionalidade do comando `while` (ou seja, os comandos que implementam a repetição de comandos até que a expressão seja falsa).

Os comandos que precedem o `while` (os pontilhados das linha 3 do Algoritmo 2.6) são traduzidos na sequência em que aparecem, e estão representados na linha 4 do Algoritmo 2.7. De forma análoga se traduz os comandos de sucedem o `while`.

O início do `while` (linha 4 do Algoritmo 2.6) é traduzido para assembly com um rótulo (nomeado `while` no Algoritmo 2.7), que indica o início da construção. Em seguida, traduz-se a expressão, e o fluxo será desviado para o rótulo `fim_while` (linha 10 do Algoritmo 2.7) se o resultado da expressão for falso. O desvio da linha 7 (`jumpSeFalso`) deve ser substituído pelo comando de desvio apropriado (vários deles são apresentados na página 29).

Os comandos que compõe o `while` (linha 5 do Algoritmo 2.6) são traduzidos para assembly na sequência em que aparecem. Após o último destes comandos, a tradução apresenta uma instrução de desvio incondicional (linha

```

1  .section .text
2  .globl _start
3  _start:
4      ...
5  while:                                # rotulo de inicio da repeticao
6      ...                              # Traducaao da Expressao (E)
7      jumpSeFalso fim_while
8      ...                              # Traducaao dos Comandos C1, C2, ..., Cn
9      jmp while                         # rotulo de saida da repeticao
10 fim_while:
11     ...

```

Algoritmo 2.7 – Tradução do comando while da Figura 2.6 para assembly

9 do Algoritmo 2.7) para o rótulo while (linha 5). Isto significa que, em tempo de execução, a instrução seguinte a ser executada é aquela contida no rótulo while (tradução da expressão).

Como exemplo de tradução de um programa completo, considere os Algoritmos 2.8 e 2.9.

```

1  long int i, a;
2  int main ( long int argc, char **argv)
3  {
4      i=0;
5      a=0;
6      while ( i<10 )
7      {
8          a+=i;
9          i++;
10     }
11     return (a);
12 }

```

Algoritmo 2.8 – Programa whileC.c

```

1  .section .data
2      I: .quad 0                # mapeado em %rax
3      A: .quad 0                # mapeado em %rdi
4  .section .text
5  .globl _start
6  _start:
7      movq $0, I
8      movq $0, A
9      movq I, %rax
10 while:
11     cmpq $10, %rax
12     jge fim_while
13     movq A, %rdi
14     addq %rax, %rdi
15     movq %rdi, A
16     addq $1, %rax
17     movq %rax, I
18     jmp while
19 fim_while:
20     movq $60, %rax
21     syscall

```

Algoritmo 2.9 – Tradução do programa whileC.c (Algoritmo 2.8) para assembly

Há dois aspectos importantes a serem levantados no programa traduzido (Algoritmo 2.9): o desvio para fora do laço e o acesso às variáveis.

O primeiro aspecto é o desvio para fora do laço. O comando `while (i<10)` significa que o laço deve ser interrompido quando a condição do laço for falsa. Em assembly, a instrução de desvio apropriada é algo como "desvie se $i \geq 10$ " (ou seja, desvie se a expressão $(i < 10)$ for falsa). Isto é representado com a instrução genérica `jumpSeFalso`, que neste caso é `jge`.

O segundo aspecto, acesso às variáveis, refere-se a como as variáveis globais "i" e "a" do programa em linguagem C foram mapeados para os rótulos "I" e "A" no programa assembly.

Quando ocorre uma atribuição a alguma variável de um programa de alto nível, esta atribuição é mapeada para o endereço da variável correspondente (neste caso, variável global). Porém, como o acesso à memória é mais lento do que o acesso a registradores, é mais eficiente mapear as variáveis em registradores para minimizar os acessos à memória.

Desta forma, podemos construir um programa equivalente ao programa contido no Algoritmo 2.9 ao considerar que "i" está mapeado no registrador `%rax` e que "a" está mapeado no registrador `%rdi`. Desta forma, podemos eliminar as linhas 2, 3, 7, 8, 9, 13, 15 e 17, para obter o Algoritmo 2.10. Este programa é equivalente ao programa do Algoritmo 2.9, porém como ele só acessa registradores, é mais rápido⁷.

```

1  .section .text
2  .globl _start
3  _start:
4      movq $0, %rax           # %rax contém I
5      movq $0, %rdi          # %rdi contém A
6  while:
7      cmpq $10, %rax
8      jge fim_while
9      addq %rax, %rdi
10     addq $1, %rax
11     jmp while
12 fim_while:
13     movq $60, %rax
14     syscall

```

Algoritmo 2.10 – Tradução do programa `whileC.c` (Algoritmo 2.8) sem acessos à memória

Um dos desafios da tradução de programas escritos em linguagens de alto nível para programas em linguagem assembly é maximizar a quantidade de variáveis mapeadas em registradores, e com isso melhorar o desempenho do programa. Os compiladores são capazes de fazer esta tarefa muito bem, porém o resultado final depende muito da quantidade de registradores que estão disponíveis na arquitetura alvo. Por esta razão, a maior parte dos processadores modernos são projetados com um grande número de registradores, uma vez que quando o número de registradores é pequeno a tendência é que programas que usam muitas variáveis tenham um desempenho inferior daquele obtido em processadores com muitos registradores. É importante observar que é cada vez mais incomum encontrar programas úteis que usam poucas variáveis.

Uma solução frequente para melhorar o desempenho dos processadores é anexar uma memória super-rápida, bem próxima à CPU (ou até dentro), chamada de memória *cache*, cuja capacidade de armazenamento de dados é muito superior à capacidade de armazenamento de dados do processador e o tempo de acesso não é muito maior do que o acesso a registradores. Porém o inconveniente é o custo, que é **muito** superior à memória convencional. Por esta razão, as memórias cache são normalmente pequenas.

2.5 Comandos Condicionais

As linguagens de programação apresentam normalmente pelo menos dois tipos de comandos condicionais: `if-then` e `if-then-else`. Se a expressão testada for verdadeira, a sequência de comandos contida nos comandos assembly relativos ao "then" deve ser executada, enquanto que se a condição for falsa, a sequência de comandos do "else" deve ser executada (se houver "else", obviamente). Os comandos do "then" e "else" não podem ser executados consecutivamente - ou executa um ou outro, mas nunca os dois.

O modelo de um comando condicional em C é apresentado no Algoritmo 2.11 e o arquivo assembly correspondente é apresentado no Algoritmo 2.12.

⁷ neste caso, algo como 10^{-9} segundos mais rápido.


```

1  int  main ( long int argc, char **argv)
2  {
3      ...
4      if ( E )
5      {
6          C_then-1, C_then-2, ... C_then-n;
7      }
8      else
9      {
10         C_else-1, C_else-2, ... C_else-m;
11     }
12     ...
13 }

```

Algoritmo 2.11 – Comando If-Then-Else (Linguagem de alto nível)

```

1  .section .text
2  .globl _start
3  _start:
4      ...           # Tradução da Expressão (E)
5      jumpSeFalso else
6      ...           # Tradução dos Comandos do then (C_then-1, C_then-2, ... C_then-n; )
7      jmp fim_if
8  else:
9      ...           # Tradução dos Comandos do else (C_else-1, C_else-2, ... C_else-m; )
10 fim_if:
11 ...

```

Algoritmo 2.12 – Tradução do comando if-then-else do Algoritmo 2.11 para assembly

A tradução segue os moldes apresentados na estrutura repetitiva. A execução depende do resultado da expressão. Se a expressão for verdadeira, a sequência de instruções incluirá as linhas 6, 7 e 10. Se a expressão for falsa, a sequência seguirá as linhas 8, 9 e 10. No primeiro caso, serão executados os comandos “then” e no segundo os comandos relativos a “else”. Não há nenhuma forma de se executar os comandos relativos a “then” e “else”, como manda o comando if.

Como exercício, traduza o Algoritmo 2.13 para assembly, monte e execute o programa através do gdb. Em seguida, altere os valores das variáveis para que a execução siga a outra alternativa. A ideia com este exercício é se familiarizar com todo o processo de geração do programa executável, com o fluxo de execução, e principalmente com o modelo de memória virtual que é adotada no linux (endereço dos rótulos da seção .data e da seção .text, endereço de início do programa, etc.).

```

1
2  long int a, b;
3  int  main ( long int argc, char **argv)
4  {
5      a=4; b=5;
6      if (a>b)
7      {
8          a=a+b;
9      }
10     else
11     {
12         a=a-b;
13     }
14     return a;
15 }

```

Algoritmo 2.13 – Exemplo para tradução

2.6 Vetores

Para finalizar o capítulo, esta seção apresenta um exemplo que combina comandos condicionais e repetitivos. Para aprofundar o conhecimento da seção de dados, este exemplo utiliza um vetor e descreve as formas de acessar este tipo de informação na memória virtual.

O programa é apresentado no Algoritmo 2.14, e contém um vetor (fixo) de dados. Este programa retorna em `$?` o valor do maior elemento do vetor. A variável `maior` armazena sempre este valor a cada iteração.

O vetor não tem um número fixo de elementos. Para determinar o fim do vetor usa-se um “sentinela”, ou seja, um elemento do vetor com valor fixo (no caso, zero). Além disso, pressupõe-se que existe pelo menos um elemento no vetor além do sentinela.

```

1 long int data_items[]={3, 67, 34, 222, 45, 75, 54, 34,
2                          44, 33, 22, 11, 66, 0};
3 long int i, maior;
4 int main (long int argc, char **argv)
5 {
6     maior = data_items[0];
7     i=1;
8     while (data_items[i] != 0)
9     {
10         if (data_items[i] > maior)
11             maior = data_items[i];
12         i++;
13     }
14     return (maior);
15 }

```

Algoritmo 2.14 – Arquivo maiorC.c

A tradução deste programa é apresentada no Algoritmo 2.15, e as novidades são:

1. a forma de criar um vetor com valores fixos. Basta listá-los lado a lado na seção `data`, ao lado do rótulo associado àquele vetor (veja a linha 6 do Algoritmo 2.15);
2. a forma de referenciar cada elemento do vetor: `movq data_items(, %rdi, 8), %rbx`. Este comando usa uma forma de endereçamento especial, chamado “endereço indexado” (veja Apêndice B.6).

Observe que o código assembly declara as variáveis `MAIOR` e `I`, porém não as acessa na memória mas sim nos registradores onde foram mapeados, ou seja, `%rbx` e `%rdi` respectivamente.

Neste ponto é interessante explicar o mapeamento das variáveis declaradas na seção `.data` na memória virtual (Figura 2).

Considere o programa assembly do Algoritmo 2.15. Ele contém vários rótulos na seção `data`. Estes rótulos indicam o nome das variáveis mapeadas na memória. Conforme descrito no Apêndice C, o depurador `gdb` permite o uso do comando `print` para visualizar o conteúdo da memória indicada por um rótulo.

Como exemplo, veja o resultado da aplicação deste comando no `gdb` tomando como alvo o programa do Algoritmo 2.15.

```

$ gdb maior
(gdb) break fim_loop
(gdb) run
(gdb) print (long)DATA_ITEMS
$1 = 3
(gdb) print (long)DATA_ITEMS+8
$2 = 11
(gdb) print &DATA_ITEMS
$3 = (<data variable, no debug info> *) 0x6000fe
(gdb) print &_start
$4 = (<text variable, no debug info> *) 0x4000b0 <_start>
(gdb) x/8 0x6000fe
0x6000fe:      0x0000000000000003      0x0000000000000043

```

```

1  .section .data
2  I:      .quad 0          # Mapeado em %rdi
3  MAIOR:  .quad 0          # Mapeado em %rbx
4                                     # %rax armazena DATA_ITEMS[i]
5  DATA_ITEMS: .quad 3, 67, 34, 222, 45, 75, 54, 34, 44, 33, 22, 11, 66, 0
6  .section .text
7  .globl _start
8  _start:
9      movq $0, %rdi
10     movq DATA_ITEMS(, %rdi, 8), %rbx    # %rbx := DATA_ITEMS[0]
11     movq $1, %rdi
12     loop:
13         movq DATA_ITEMS(, %rdi, 8), %rax    # %rax := DATA_ITEMS[i]
14         cmpq $0, %rax
15         je fim_loop                        # se DATA_ITEMS[i] == 0, sai
16         cmpq %rbx, %rax
17         jle fim_if                        # se DATA_ITEMS[i] <= MAIOR, sai do "if"
18         movq %rax, %rbx                    # MAIOR = DATA_ITEMS[i]
19         fim_if:
20             addq $1, %rdi                    # i := i+1
21             jmp loop
22     fim_loop:
23         movq %rbx, %rdi
24         movq $60, %rax
25         syscall
26

```

Algoritmo 2.15 – Arquivo maiorS.s

```

0x60010e:      0x0000000000000022      0x00000000000000de
0x60011e:      0x000000000000002d      0x000000000000004b
0x60012e:      0x0000000000000036      0x0000000000000022
(gdb) x/8d 0x6000fe
0x6000fe:      3          67
0x60010e:      34          222
0x60011e:      45          75
0x60012e:      54          34

```

Observe que:

- o comando `print (long)DATA_ITEMS` imprime $M[DATA_ITEMS]$ como visto como se fosse um `long int`. Esta coerção é necessária pois `DATA_ITEMS` foi declarado como um `.quad` (uma palavra quádrupla) que não está associado a nenhum tipo de variável;
- o comando `print (long)DATA_ITEMS+8` imprime $M[DATA_ITEMS] + 8$ (e não $M[DATA_ITEMS + 8]$);
- o comando `print &DATA_ITEMS` imprime o endereço de `DATA_ITEMS` em tempo de execução (que é `0x6000fe`);
- o comando com o comando `print &_start` mostra que é possível imprimir o endereço de qualquer rótulo (não só os da seção `.data`);
- o comando `x/8 0x6000fe` imprime oito palavras quádruplas a partir do endereço `0x6000fe`. Saída é em hexadecimal;
- o comando `x/8d 0x6000fe` imprime oito palavras quádruplas a partir do endereço `0x6000fe`. A impressão converte cada palavra quádrupla em inteiro longo.

Exercícios

- 2.1 Proponha uma forma de tradução para os comandos “for”, “switch” e “do .. while”.
- 2.2 Desenhe uma figura semelhante à Figura 2 detalhando onde é mapeada a seção `text` e a seção `data`.
- 2.3 Altere o algoritmo 2.15 para ordenar o vetor.

- 2.4 O endereço inicial da seção `.data` que aparece nos exemplos é próximo de `0x0060 0000` enquanto que o endereço inicial da seção `.text` é próximo de `0x0040 0000`. Isso sugere que o programa executável pode ocupar, no máximo, `0x0020 0000` bytes. O que ocorre se a seção `.text` for maior do que `0x0020 0000` bytes?
- 2.5 Escreva um programa que imprime o tamanho das suas seções `.text` e `.data`.

3. A Seção Pilha

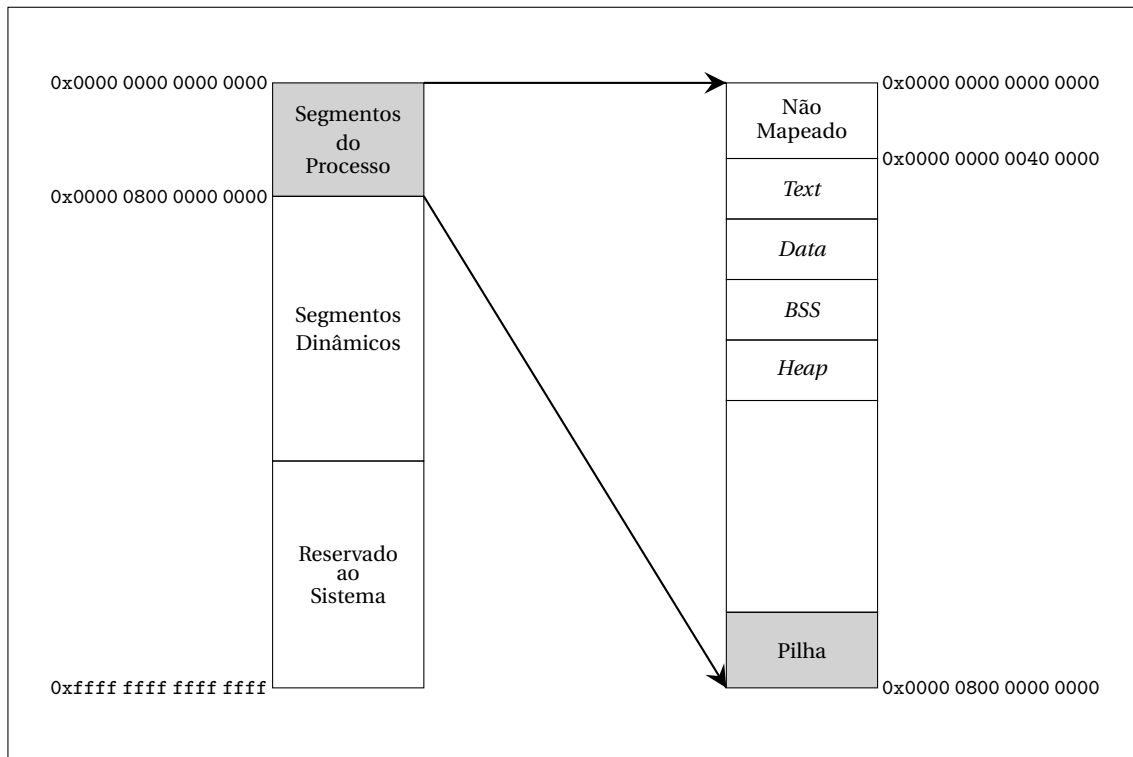


Figura 4 – Espaço Virtual de um Processo: Seção *Stack*

A seção pilha (*stack*) armazena informações sobre chamadas de procedimento (parâmetros, informações sobre contexto e variáveis locais).

A ideia de armazenar estas informações em uma pilha é originária da linguagem de programação Algol 60¹. Antes de Algol 60, as linguagens de programação só permitiam o uso de variáveis globais, sem procedimentos. Exponentes desta época foram as linguagens Fortran e Cobol, que posteriormente foram adaptadas para comportar procedimentos e ainda mais tarde, variáveis locais.

Antes de detalhar o tema, é necessário compreender o que ocorre em uma chamada de procedimento, ou seja, o modelo utilizado para implementar chamadas de procedimento em praticamente todas as linguagens de programação atuais.

A Figura 4 destaca a parte do programa em execução que será abordado neste capítulo, que corresponde ao intervalo entre o endereço virtual 0x0000 0800 0000 0000 e um limite variável dado pelo valor atual do registrador `%rsp`.

O capítulo está organizado da seguinte forma: a Seção 3.1 enumera as informações que devem ser preservadas em uma chamada de procedimentos. Estas informações são salvas em uma estrutura denominada “registro de ativação”, e a Seção 3.2 descreve passo a passo como criar um registro de ativação em uma arquitetura genérica, porém exemplificando com o *assembly* do AMD64. Por fim, a Seção 3.3 apresenta aspectos práticos do AMD64, da linguagem C e de segurança.

¹ A ideia aparentemente é oriunda dos estudos do pesquisador alemão Friedrich L. Bauer, que também trabalhou no desenvolvimento da linguagem Algol 60, mais especificamente no uso da pilha para cálculos de expressões aritméticas e lógicas. Ele até ganhou um prêmio de pioneirismo da IEEE (IEEE Computer Society Pioneer Award) em 1988 pelo seu trabalho.

3.1 Informações Armazenadas nas Chamadas de Procedimento

Esta seção descreve quais as informações que devem ser armazenadas nas chamadas de procedimento e por quanto tempo. Para tal, a Seção 3.1.1 descreve o escopo das variáveis de procedimentos, a Seção 3.1.2 descreve quais informações adicionais devem ser armazenadas e a Seção 3.1.3 apresenta a estrutura onde estas informações são armazenadas, o registro de ativação.

3.1.1 Escopo de Variáveis

Na maior parte das linguagens de programação é possível declarar variáveis globais (visíveis em todos os pontos do programa) e variáveis locais e parâmetros (visíveis somente no procedimento em que são declarados). Esta “visibilidade” de uma variável é chamado escopo da variável.

Em tempo de execução, as variáveis globais são sempre visíveis e as variáveis de um procedimento são alocadas quando o procedimento é chamado e desalocadas quando da saída do procedimento.

Na linguagem C, qualquer variável declarada fora dos procedimentos é uma variável global e qualquer variável declarada dentro de um procedimento é uma variável local ou parâmetro.

Vamos agora analisar o que ocorre com as variáveis locais e parâmetros de um procedimento em tempo de execução. A ideia geral é que estas variáveis devem ser alocadas na memória quando o procedimento for chamado, e desalocadas na memória quando o procedimento for finalizado.

Considere um programa onde procedimento $A()$ chama um procedimento $B()$. Em tempo de execução, as variáveis de $A()$ devem ser alocadas quando ele for chamado. Quando ele chamar $B()$, as variáveis de $B()$ devem ser alocadas, mas sem afetar as variáveis de $A()$. Quando sair do procedimento $B()$, as variáveis de $B()$ devem ser liberadas, mas sem causar nenhum efeito nas variáveis de $A()$.

O mesmo ocorre em chamadas recursivas. Considere um programa com um procedimento $R()$. Em tempo de execução, a primeira chamada a $R()$ provoca a alocação de todas as variáveis de $R()$. Vamos denominar estas variáveis de V_R . Na chamada recursiva de $R()$ (que denominaremos $R'()$) um novo conjunto de variáveis deve ser alocado, que denominaremos de $V_{R'}$.

As alterações em $V_{R'}$ não afetam V_R . Quando sair do procedimento $R'()$, V_R continua intacto.

3.1.2 Informações de Contexto

As informações de contexto são aquelas que coordenam o fluxo de execução na sequência de chamada e retorno de procedimentos e o acesso às variáveis locais e parâmetros de um procedimento.

Estas informações devem ser salvas para que um procedimento não interfira em outro, e esta seção descreve, genericamente, como fazê-lo.

Primeiro iremos descrever a sequência de chamada e de retorno de procedimentos. Quando se faz a chamada de procedimento, sabe-se que o fluxo é desviado para o procedimento. Na saída do procedimento, sabe-se o fluxo deve retornar à instrução seguinte à chamada. A questão é como fazê-lo.

O mecanismo mais simples é utilizar uma pilha. Antes de desviar o fluxo para o procedimento, armazena-se na pilha o endereço da instrução seguinte. Para retornar o fluxo de execução de volta ao chamador, utiliza-se o endereço contido no topo da pilha. Observe que este mecanismo simples permite a chamada de vários procedimentos e os respectivos retornos sem haver confusão.

O passo seguinte é descrever o acesso às variáveis. Para entender o problema, considere mais uma vez um programa onde procedimento $R()$ é recursivo. Na primeira chamada, as variáveis de $R()$, que chamaremos de V_R , devem ser alocadas e podem ser acessadas. Seja v_r uma variável declarada em V_R . Quando o procedimento $R()$ acessar v_r , a instância a ser usada é $v_r \in V_R$.

Agora considere as chamadas recursivas de $R()$ que denominaremos $R'()$, $R''()$, e assim por diante. Sejam $V_{R'}$ as variáveis de $R'()$, $V_{R''}$ as variáveis de $R''()$, e assim por diante.

Agora considere que o procedimento $R()$ acessa v_r . Existem dois v_r : $v_r \in V_{R'}$ e $v_r \in V_R$, sendo que $V_R \neq V_{R'}$. O correto é acessar a última alocada, ou seja, $v_r \in V_{R'}$.

Vamos agora nos concentrar no que ocorre quando o procedimento $R'()$ finalizar. O espaço alocado para $V_{R'}$ deve ser liberado e o fluxo retorna à instrução seguinte à chamada, dentro de $R()$. Neste ponto considere que $R()$ volta a acessar v_r . Neste caso, deve acessar $v_r \in V_R$.

Para atingir este objetivo, uma solução é usar uma variável global que indica qual o conjunto de variáveis deve ser usada a cada momento (no exemplo, V_R ou $V_{R'}$). Seja r_{base} esta variável.

Assim como no caso do endereço de retorno, é necessário salvar o valor corrente de r_{base} antes de alterar o seu valor. Novamente, uma pilha é uma estrutura adequada.

Voltando ao exemplo, considere que em $R()$, temos $r_{base} \rightarrow V_R$ (r_{base} aponta para V_R). Quando acessar v_r , usaremos $v_r[r_{base}]$ (ou seja, $v_r \in V_R$). Ao chamar $R()$ recursivamente, salvamos r_{base} na pilha e o substituímos por $r_{base} \rightarrow V_{R'}$. Observe que o acesso a v_r é exatamente igual, $v_r[r_{base}]$, só que agora $v_r \in V_{R'}$.

Ao retornar do procedimento, restaura-se o valor de r_{base} do topo da pilha, ou seja, $r_{base} \rightarrow V_R$. O acesso a v_r continua igual: $v_r[r_{base}]$, só que agora $v_r \in V_R$.

Existem várias formas de implementar este mecanismo. Normalmente, a variável r_{base} é armazenada em um registrador (que só é usado para isso) para ganhar desempenho. Este registrador recebe nomes diferentes dependendo da CPU. Por exemplo, no AMD64 ele é chamado *base register*, registrador de base (%rbp) enquanto no MIPS ele é chamado *frame pointer* (fp).

Apesar de tecnicamente correta, a explicação acima é muito complicada para ser entendida por iniciantes. Por esta razão, as próximas 15 ou 20 páginas detalham na prática o que se explicou nas últimas 35 linhas.

3.1.3 Registro de Ativação

“Registro de Ativação” é o termo adotado para indicar a estrutura criada em tempo de execução para armazenar:

1. variáveis de um procedimento:
 - a) variáveis locais;
 - b) parâmetros;
2. informações de contexto:
 - a) endereço de retorno;
 - b) registrador de base.

Conforme explicado nas Seções 3.1.1 e 3.1.2, todas estas informações devem ser armazenadas em uma pilha.

Por esta razão, o registro de ativação também funciona como uma pilha. O local específico onde ela é alocada e liberada está na região *stack*². Veja a região destacada na Figura 4).

O modelo esquemático de um registro de ativação é apresentado na Figura 5.

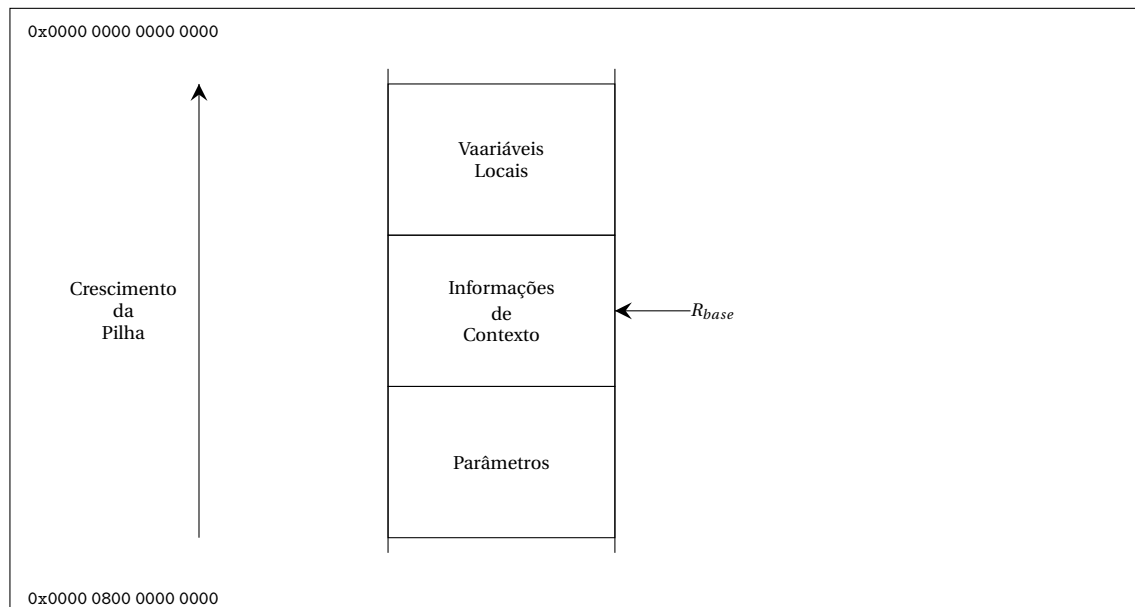


Figura 5 – Registro de ativação: modelo esquemático

A ordem em que as informações são armazenadas é importante. Para entender por que, considere uma chamada de procedimento, digamos $P(p1, p2)$.

² pilha em português

1. Antes de desviar para o procedimento P, empilham-se os parâmetros (p1 e p2).
2. Executa-se a chamada de procedimento salvando o endereço de retorno na pilha (na área das informações de contexto).
3. Salva-se o valor corrente do registrador de base (r_{base}), e substitui o valor de r_{base} para que ele aponte para as variáveis do registro de ativação corrente.
4. Por fim, aloca espaço para as variáveis locais.

O registrador de base (r_{base}) aponta sempre para um mesmo local no registro de ativação (mas não para o mesmo endereço de memória). Como a quantidade de informações de contexto é fixa, basta somar uma constante a r_{base} para acessar as variáveis locais, e basta subtrair uma constante para acessar os parâmetros.

Vamos detalhar melhor o que significa dizer que os registros de ativação seguem o modelo de pilha com um exemplo. Quando ocorre a chamada de um procedimento A(), o registro de ativação de A(), digamos $RA_{A()}$ é empilhado, e fica no topo da pilha. Se A() chamar um procedimento B(), um novo registro de ativação para B(), digamos $RA_{B()}$ é empilhado sobre $RA_{A()}$ e $RA_{B()}$ é que estará no topo da pilha. Quando B() finalizar, $RA_{B()}$ é desempilhado e $RA_{A()}$ estará no topo da pilha. É importante destacar que r_{base} sempre aponta para o registro de ativação do topo da pilha.

3.2 Implementação do Modelo em Uma Arquitetura

O modelo genérico apresentado na seção anterior é implementado de formas diferentes dependendo do conjunto linguagem de programação/sistema operacional/CPU. Esta seção descreve uma versão simplificada da implementação descrita em [51] para o conjunto linguagem C/Linux/AMD64. Esta versão simplificada só trata de parâmetros inteiros de 64 bits (`long int` na linguagem C).

O primeiro aspecto importante a ser lembrado é que os endereços das variáveis e dos procedimentos são definidas em tempo de ligação. Os endereços dos procedimentos podem ser fixos (são associados a rótulos), mas o mesmo não pode ser feito para as variáveis.

Para fins didáticos, esta seção mostra como alocar e liberar registros de ativação em várias etapas. A Seção 3.2.1 descreve os registros de ativação sem parâmetros e sem variáveis locais. A Seção 3.2.2 acrescenta as variáveis locais ao modelo da seção anterior e a Seção 3.2.3 acrescenta os parâmetros passados por valor. A Seção 3.2.4 descreve parâmetros passados por referência. Para finalizar, a Seção 3.2.5 apresenta um exemplo de um procedimento recursivo, que usa todos os conceitos apresentados.

3.2.1 Sem Parâmetros e sem Variáveis Locais

Veja novamente a Figura 5. Se eliminarmos o espaço destinado às variáveis locais e aos parâmetros, o que sobra é o espaço destinado às informações de contexto. Ao contrário do espaço reservado a variáveis e parâmetros (que podem não existir), o espaço destinado às informações de contexto **SEMPRE** é alocado quando ocorre uma chamada de procedimento.

Conforme descrito na Seção 3.1.3, o registro de ativação armazena duas informações: o endereço de retorno e o registrador de base.

A Figura 6 indica como estas informações devem ficar armazenadas no espaço reservado às informações de contexto. A figura já contém os registradores sugeridos em [51]: `%rbp` para o registrador de base e `%rip` para o apontador de instruções.

- O valor anterior de `%rbp` é armazenado na pilha. Após armazená-lo, muda-se o valor de `%rbp` para que ele contenha o endereço onde está armazenado o valor anterior.
- O endereço de retorno é o endereço da instrução seguinte à chamada do procedimento. Este endereço está armazenado no registrador `%rip`.

Antes de apresentar um exemplo de como construir o registro de ativação, vamos abordar um aspecto que omitimos até o momento: quem indica o topo da *stack* e o que isto significa.

A Figura 4 mostra a região *stack* em uma área destacada. O que a figura não mostra é o que determina o final desta região.

Em todas as CPUs modernas (que eu conheço), um registrador é utilizado para indicar o limite superior da *stack*. No AMD64, este registrador é o `%rsp`³. O seu conteúdo é o endereço do limite superior da *stack*. Em outras palavras, o programa em execução pode acessar qualquer endereço entre 0x0000 0800 0000 0000 (valor superior) e o valor corrente de `%rsp` (valor inferior). Por exemplo, a instrução

³ *Stack Pointer*

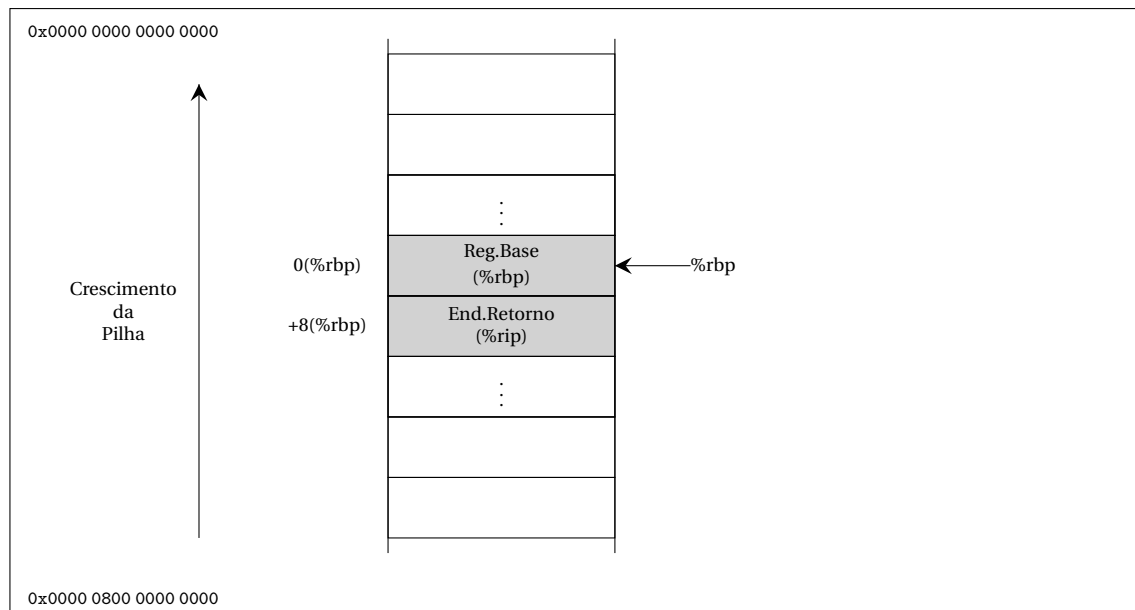


Figura 6 – Registro de ativação: Informações de Contexto

```
movq $1, 12(%rsp)
```

acessa o conteúdo de um endereço de memória 12 bytes abaixo de `%rsp`. Como este endereço está entre `%rsp` e o endereço `0x0000 0800 0000 0000` (a área cinza da Figura 4), o acesso é válido.

Por outro lado, a instrução

```
movq $1, -12(%rsp)
```

tenta acessar o conteúdo de um endereço de memória 12 bytes acima de `%rsp`, uma região não válida. Neste caso, o acesso é inválido e deveria gerar um erro. Curiosamente, o programa pode indicar erro ou não, dependendo do mapeamento das páginas na memória virtual (veja o Apêndice D para uma explicação da memória virtual e em especial do mecanismo conhecido como paginação).

O registrador `%rsp` é tão importante que existem instruções especiais que o usam implicitamente para empilhar (`pushq`) e desempilhar (`popq`). A funcionalidade das duas é descrita na Tabela 3 onde se vê que (1) a instrução `pushq` abre um espaço no topo da pilha e lá armazena o conteúdo do parâmetro indicado e que (2) a instrução `popq` faz o inverso: armazena o topo da pilha no parâmetro indicado e libera o espaço no topo da pilha.

<code>pushq <parâmetro></code>	<code>subq \$8, %rsp</code> <code>movq <parâmetro>, (%rsp)</code>
<code>popq <parâmetro></code>	<code>movq (%rsp), <parâmetro></code> <code>addq \$8, %rsp</code>

Tabela 3 – Funcionalidade das instruções `pushq` e `popq`

Exemplos:

```
.section .data
A: .quad 10
.section .text
movq $11, %rax # armazena a constante 11 em %rax
pushq $1       # empilha a constante 1
pushq $A       # empilha o endereço do rótulo A
pushq A        # empilha o conteúdo do rótulo A
```

```

pushq %rax    # empilha o conteúdo do registrador %rax
popq $1       # ERRO! O destino não é um local de escrita
popq $A       # ERRO! O destino não é um local de escrita
popq A        # desempilha o topo da pilha para o endereço indicado pelo rótulo A
popq %rax     # desempilha o topo da pilha no registrador %rax

```

Para exemplificar como montar o registro de ativação neste caso, considere o Algoritmo 3.1, traduzido para *assembly* no Algoritmo 3.2.

```

1 long int a, b;
2
3 long int soma ( )
4 {
5     return (a+b);
6 }
7
8 int main (long int argc, char **argv)
9 {
10     a=4;
11     b=5;
12     b = soma();
13     return (b);
14 }

```

Algoritmo 3.1 – Programa sem parâmetros e sem variáveis locais

```

1
2 .section .data
3     A: .quad 0
4     B: .quad 0
5 .section .text
6 .globl _start
7 soma:
8     pushq %rbp                # empilha %rbp
9     movq %rsp, %rbp
10    movq A, %rax
11    movq B, %rbx
12    addq %rax, %rbx
13    movq %rbx, %rax
14    popq %rbp                 # desempilha em %rbp
15    ret                       # desempilha em %rip
16 _start:
17    movq $4, A
18    movq $5, B
19    call soma                  # empilha %rip / jmp soma
20    movq %rax, %rbx
21    movq $60, %rax
22    movq %rbx, %rdi
23    syscall

```

Algoritmo 3.2 – Tradução do programa do Algoritmo 3.1

O Algoritmo 3.2 apresenta instruções novas.

O primeiro par de instruções, `pushq` (linha 8) e `popq` (linha 14), que já foram explicados, salvam e restauram o valor de `%rbp` na pilha.

O segundo par de instruções é `call` (linha 19) e `ret` (linha 15). Elas são parecidas com o primeiro par, porém usam o registrador `%rip` (instruction pointer) como parâmetro implícito. A Tabela 4 apresenta o esquema de funcionamento deste segundo par utilizando `pushq` e `popq` para efeitos didáticos.

A instrução `call <rotulo>` primeiro empilha o valor de `%rip` (ou seja, o endereço da próxima instrução) e depois desvia o fluxo para o rótulo indicado. Por outro lado, a instrução `ret` armazena no registrador `%rip` o valor contido no topo da pilha.

call <rotulo>	pushq %rip jmp <rotulo>
ret	popq %rip

Tabela 4 – Funcionalidade das instruções call e ret

O leitor deve estar se perguntando por que foi criada uma instrução (ret) se ela é equivalente à uma instrução popq já existente. Parece desnecessário.

A resposta está na segurança. Os primeiros computadores permitiam que qualquer instrução lesse e escrevesse no registrador que indica a próxima instrução (no AMD64, é o %rip). Porém, a experiência com erros de programadores descuidados e mal intencionados levaram os fabricantes de computador a restringir o acesso a este registrador.

Por esta razão, o AMD64 (como todas as CPUs que conheço) não permite leitura ou escrita diretas no registrador %rip. Por exemplo, as instruções abaixo são inválidas:

```
movq %rax, %rip    # Instrução inválida: destino não pode ser %rip
movq %rip, %rax    # Instrução inválida: origem não pode ser %rip
```

Porém, existem formas de contornar esta proibição (veja Exercício 3.1).

Voltando ao Algoritmo 3.2, observe a instrução call soma que está na linha 19. Ela armazena o valor de %rip (que aponta para a instrução da linha 20, movq %rax, %rbx) no topo da pilha e depois desvia o fluxo para o rótulo soma.

A última instrução do procedimento soma é a instrução ret (linha 15), que coloca em %rip o valor contido no topo da pilha, que é o endereço da instrução movq %rax, %rbx na linha 20.

Neste ponto o leitor deve estar se perguntando como é que se tem certeza que nesta hora o valor no topo da pilha é o endereço de retorno (afinal, muita coisa pode ser empilhada e mudar o valor de %rsp). A ideia é que qualquer informação colocada na pilha deve ser desempilhada em ordem invertida. Seguindo o fluxo de execução, temos:

```
call      # linha 19
pushq %rbp # linha 8
popq %rbp  # linha 14
ret       # linha 15
```

E aqui fica mais claro o porquê da certeza da instrução ret encontrar o endereço de retorno no topo da pilha ao ser executada.

3.2.2 Sem Parâmetros e com Variáveis Locais

O acesso a variáveis locais e parâmetros de um registro de ativação se dá através do uso de um registrador que sempre aponta para o mesmo local do registro de ativação. No caso do AMD64, este registrador é o %rbp.

A Figura 7 mostra um registro de ativação com informações de contexto e com n variáveis locais já alocadas. Para simplificar, consideramos que todas as variáveis são inteiros e por isso ocupam oito bytes.

As variáveis locais ficam localizadas imediatamente acima do local apontado por %rbp. Como ele sempre aponta para o mesmo local da moldura do registro de ativação, é possível determinar o endereço de cada variável: a primeira variável localiza-se em $-8(\%rbp)$. A segunda em $-16(\%rbp)$ e assim por diante. Observe o sinal de menos à frente dos números. Como a pilha cresce para cima, é natural pensar que o sinal deveria ser positivo e não negativo. Porém, a pilha cresce para cima em direção aos endereços menores, ou seja, de $0x0000\ 0800\ 0000\ 0000$ para $0x0000\ 0000\ 0000\ 0000$ conforme representado na Figura 7.

Este modo de determinar o endereço de variáveis é diferente do adotado para as variáveis globais na seção .data, onde o endereço de cada variável é fixo. O nome dado ao último é endereçamento estático e ao primeiro é endereçamento relativo (neste caso relativo a %rbp).

Com o endereçamento relativo basta, após criar um novo registro de ativação, ajustar o valor de %rbp para acessar as variáveis do novo registro de ativação.

Por fim, vamos considerar o mapeamento das variáveis nos espaços reservados a elas. Considere o programa C apresentado no Algoritmo 3.3.

As variáveis, a e b são globais e consequentemente serão alocadas na seção .data. Já as variáveis x e y são locais ao procedimento soma.

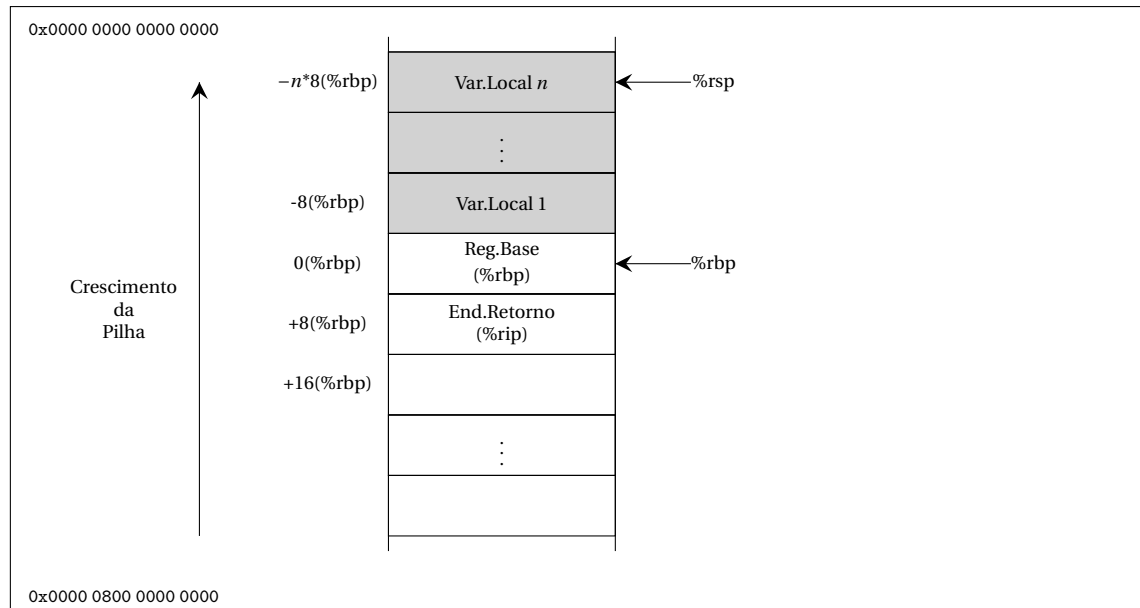


Figura 7 – Registro de ativação: Variáveis Locais

```

1 long int a, b;
2
3 long int soma ( )
4 {
5     long int x, y;
6     x=a;
7     y=b;
8     return (x+y);
9 }
10
11 int main (long int argc, char **argv)
12 {
13     a=4;
14     b=5;
15     b = soma();
16     return (b);
17 }

```

Algoritmo 3.3 – Programa sem parâmetros e com variáveis locais

A primeira variável encontrada, *x*, ocupará o espaço logo acima do apontado por *%rbp*, ou seja, $-8(\%rbp)$. Quando o programa for traduzido para *assembly*, e esta será a forma de referenciar *x*.

A tabela de mapeamento (Tabela 5) é bastante útil:

Com esta tabela, fica mais fácil memorizar como referenciar as variáveis no código assembly equivalente apresentado no Algoritmo 3.4.

A tradução da maior parte do programa segue o esquema visto até o momento. Com relação à execução, a Figura 8 descreve detalhadamente o passo a passo da construção de um registro de ativação contendo somente variáveis locais.

A Figura 8 mostra, da esquerda para a direita, cada etapa numerada de (1) até (5) da execução do programa do Algoritmo 3.4.

- (1) Situação da região *stack* antes de chegar na chamada de procedimento da linha 23. A região destacada corresponde ao registro de ativação anterior (que neste caso, não existia). Observe que o registrador *%rsp* aponta para a última posição daquele registro de ativação.
- (2) Situação imediatamente após a execução da instrução da linha 23, porém antes de executar a instrução da linha 7. A instrução *call* empilha o endereço de retorno (valor corrente de *%rip*).

Símbolo em C	Categoria	Endereço <i>assembly</i>
a	global	A
b	global	B
x	local	-8(%rbp)
y	local	-16(%rbp)

Tabela 5 – Tabela de mapeamento das variáveis do programa em C do Algoritmo 3.3 para as variáveis do programa *assembly* do Algoritmo 3.4.

```

1  .section .data
2      A: .quad 0
3      B: .quad 0
4  .section .text
5  .globl _start
6  soma:
7      pushq %rbp                # empilha (salva) %rbp
8      movq  %rsp, %rbp          # faz %rbp apontar para este R.A.
9      subq  $16, %rsp           # aloca variaveis locais
10     movq  A, %rax
11     movq  %rax, -8(%rbp)
12     movq  B, %rax
13     movq  %rax, -16(%rbp)
14     movq  -8(%rbp), %rax
15     movq  -16(%rbp), %rbx
16     addq  %rbx, %rax
17     addq  $16, %rsp           # libera variaveis locais
18     popq  %rbp               # desempilha (restaura) %rbp
19     ret                        # retorna ao chamador
20 _start:
21     movq  $4, A
22     movq  $5, B
23     call  soma
24     movq  %rax, A
25     movq  A, %rdi
26     movq  $60, %rax
27     syscall

```

Algoritmo 3.4 – Tradução do programa do Algoritmo 3.3

- (3) Situação após a execução da instrução da linha 7, que empilhou (salvou) o valor corrente de `%rbp`.
- (4) Situação após a execução da instrução da linha 8, que copia o valor contido em `%rsp` para `%rbp`. Veja que os dois registradores apontam para o mesmo endereço.
- (5) Situação após a execução da instrução da linha 9, que abre espaço para as duas variáveis locais. Isto conclui a construção do registro de ativação do procedimento `soma()`.

A Figura 9 mostra o processo inverso: o que ocorre na *stack* quando o programa do Algoritmo 3.4 desmonta o registro de ativação ao finalizar o procedimento.

O estágio inicial da Figura 9 repete o último estágio da Figura 8.

- (6) A altura da pilha (o valor de `%rsp`) é a mesma da linha 10 até a linha 16 quando as variáveis locais são liberadas.
- (7) Situação após liberar as variáveis locais, o que é feito pela instrução da linha 17. Observe que após esta instrução, a pilha fica no mesmo estado do encontrado no item estágio (4) da Figura 8.
- (8) Situação após executar a instrução da linha 18, que restaura o valor de `%rbp` para o anterior à chamada de procedimento. A Figura não mostra onde está localizado `%rbp`, que está apontando para o registro de ativação anterior (fora do campo de visão). Observe que após esta instrução, a pilha fica no mesmo estado do encontrado no estágio (3) da Figura 8.
- (9) Situação após executar a instrução da linha 19, que atribui o valor contido no endereço apontado por `%rsp` para `%rip`, ou seja, restaurando o fluxo de execução para a instrução seguinte àquela da chamada do procedimento (linha 24).

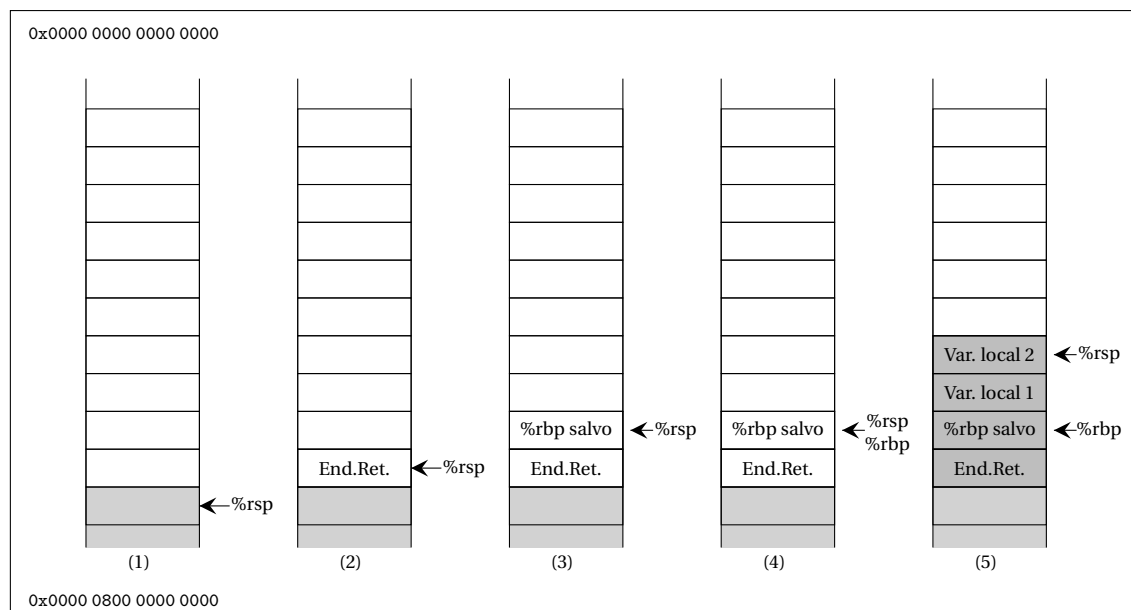


Figura 8 – Montagem do registro de ativação com variáveis locais

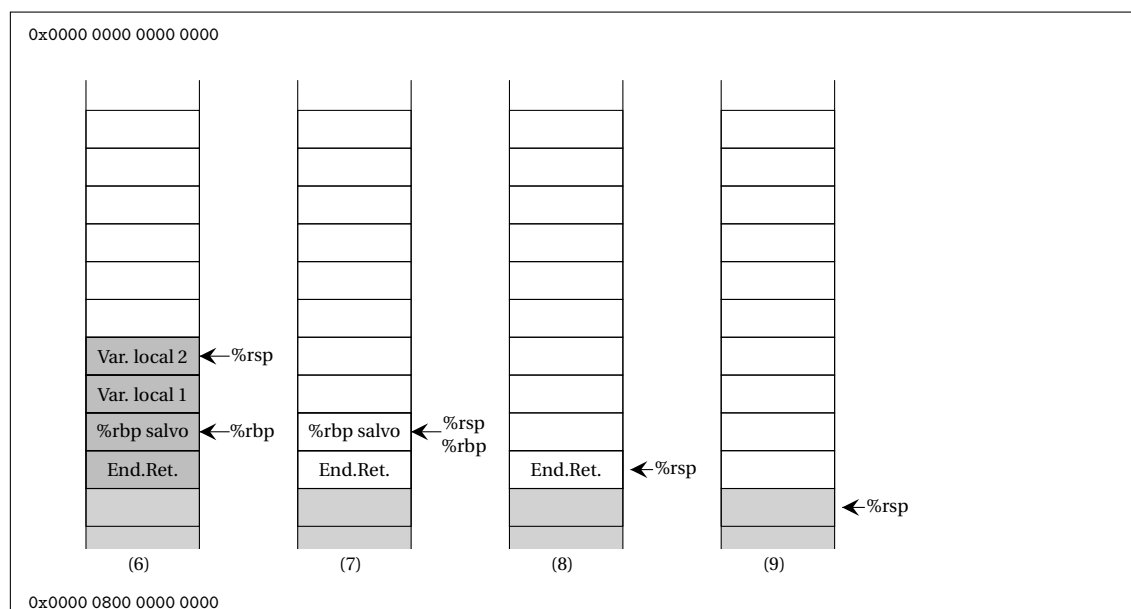


Figura 9 – Desmontagem do registro de ativação com variáveis locais

Observe que as instruções *assembly* de montagem e desmontagem do registro de ativação são complementares (Tabela 6).

3.2.3 Com Parâmetros e com Variáveis Locais

Considere o procedimento P, que tem três parâmetro inteiros e que retorna um valor inteiro:

P(int a, long int b, long int c)

Conforme apresentado no registro de ativação esquemático na Figura 5 (página 39), os parâmetros devem ser colocados na pilha **ANTES** das informações de contexto, ou seja, antes da instrução *assembly* `call`.

Figura 8	Figura 9	
(1)	(9)	Situação antes da chamada e após o retorno do procedimento (linhas 23 e 24). A altura da pilha é o mesmo. Isto não é coincidência.
(2)	(8)	A situação imediatamente após a chamada do procedimento (linha 23) mostra o endereço de retorno no topo da pilha, que será consumido pela instrução da linha 24 que retorna o fluxo ao chamador.
(4)	(7)	Assim que entra no procedimento, são executadas duas instruções: A primeira (linha 7) salva o valor antigo de <code>%rbp</code> enquanto que a segunda (linha 8) ajusta o valor de <code>%rbp</code> para apontar para o novo registro de ativação. Ao sair do procedimento, somente uma instrução é necessária (linha 18) para restaurar o valor original de <code>%rbp</code> .
(5)	(6)	A alocação das variáveis locais é feita na linha 9 e a liberação é feita na instrução da linha 17.

Tabela 6 – Comparação das Figuras 8 e 9

Ao retornar do procedimento, a altura da pilha será exatamente igual àquela que havia ao executar a instrução `call`, ou seja, com os parâmetros ainda empilhados.

Por esta razão, ao retornar do procedimento, a primeira coisa a ser feita é desempilhar os parâmetros. Para tal, basta ajustar o valor de `%rsp`.

Agora, trataremos da ordem em que os parâmetros devem ser empilhados. A primeira ideia é empilhar na ordem em que eles aparecem. No exemplo do procedimento P acima, seria (1) empilha a, (2) empilha b e (3) empilha c.

A linguagem Pascal (entre muitas outras) faz exatamente isso, mas a linguagem C faz o contrário: (1) empilha c, (2) empilha b e (3) empilha a.

Como este livro trabalha com traduções da linguagem C, este será o modelo adotado aqui. Assim, a Figura 10 mostra como ficam organizados os parâmetros em um registro de ativação.

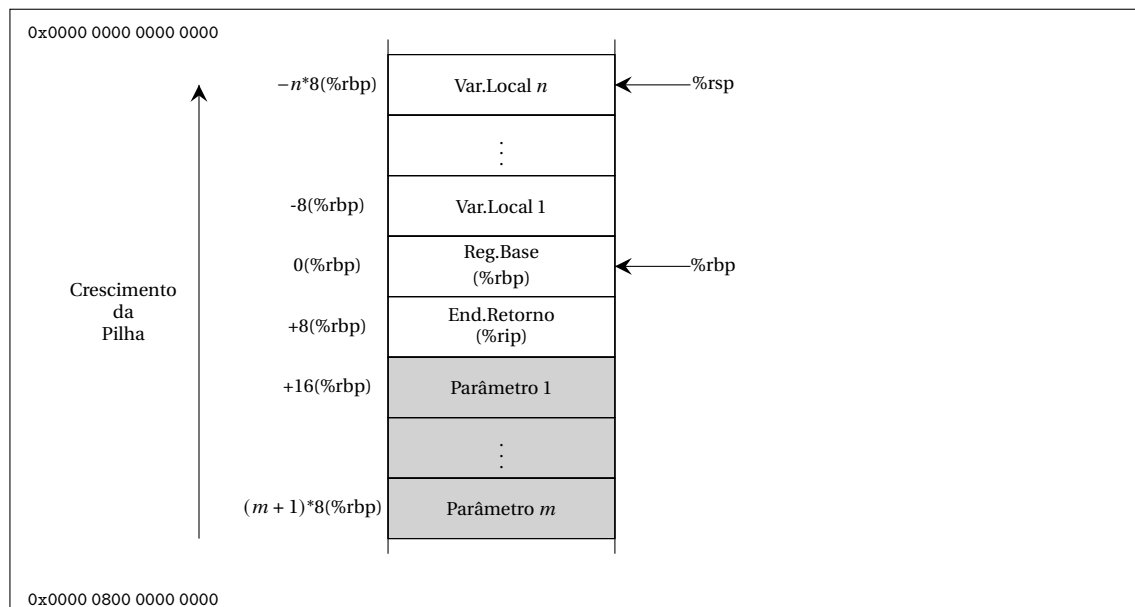


Figura 10 – Registro de ativação: Parâmetros

Um dos motivos para esta organização é permitir a criação de procedimentos com número variável de argumentos, chamados “`va_list`”.

Veja, por exemplo, as variações do comando `printf` listadas a seguir:

```
printf("um parametro")
printf("dois parametros %d", 1)
printf("três parametros %d %d ", 11, 22)
printf("quatro parametros %d %d %d ", 111, 222, 333)
printf("erro de principiante %d %d %d %d")
```

Com a organização adotada na Figura 10, os registros de ativação de todas as variações colocam o endereço do início da cadeia de caracteres no espaço reservado ao primeiro parâmetro (16(%rbp))⁴. Ao ser chamado, o procedimento `printf` analisa a cadeia de caracteres. Cada vez que encontrar o símbolo %, ela assume que haverá um parâmetro na pilha para ele. Assim, no comando

```
printf("quatro parametros %d %d %d ", 111, 222, 333)
```

o procedimento `printf` irá procurar o primeiro parâmetro (a constante 111) no local reservado ao parâmetro 2 (24(%rbp)), o segundo parâmetro (a constante 222) em 32(%rbp) e o terceiro parâmetro (a constante 333) em 40(%rbp).

Este mecanismo só funciona porque o primeiro parâmetro (a cadeia de caracteres) sempre está presente em 16(%rbp). Se os parâmetros fossem empilhados na sequência, ele estaria em posições diferentes da pilha em cada caso.

Um caso interessante é aquele em que a quantidade de parâmetros indicado no primeiro parâmetro é diferente da quantidade de parâmetros presentes, como em

```
printf("erro de principiante %d %d %d %d")
```

Neste caso, o compilador irá avisar que tem algo errado, mas o código executável será gerado. Ao executar o programa, será impresso o que estiver nos endereços onde deveriam estar os parâmetros⁵.

Para exemplificar como processo de tradução, considere o programa indicado no Algoritmo 3.5, e sua versão *assembly* no Algoritmo 3.6.

Quando for necessário eliminar a ambiguidade do termo “parâmetro”, chamaremos de **parâmetros formais** aqueles usados no procedimento (no caso, `x` e `y`) e chamaremos de **parâmetros efetivos** aqueles usados na chamada de procedimento (no caso, `a` e `b`).

```
1 long int a, b;
2 long int soma (long int x, long int y)
3 {
4     long int z;
5     z = x + y;
6     return (z);
7 }
8
9 int main (long int argc, char **argv)
10 {
11     a=4;
12     b=5;
13     b = soma(a, b);
14     return (b);
15 }
```

Algoritmo 3.5 – Programa com parâmetros e com variáveis locais

No Algoritmo 3.6, destacamos:

- os parâmetros são empilhados na ordem inversa do programa C, ou seja, primeiro empilha B e depois A (linhas 19 e 20);
- após empilhá-los é que se faz a chamada ao procedimento (linha 21);

⁴ Atenção: o endereço de início da cadeia de caracteres ocupa 8 bytes. A cadeia de caracteres é colocada em outro lugar.

⁵ É mais divertido tentar com `printf("%s")`, pois serão impressos a sequência de caracteres que inicia no endereço contido em 24(%rbp) e até encontrar um byte igual a zero (\0) o que pode demorar um pouco enchendo a tela de símbolos “curiosos”


```

1  .section .data
2      A: .quad 0
3      B: .quad 0
4  .section .text
5  .globl _start
6  soma:
7      pushq %rbp
8      movq %rsp, %rbp
9      subq $8, %rsp
10     movq 16(%rbp), %rax
11     addq 24(%rbp), %rax
12     movq %rax, -8(%rbp)
13     addq $8, %rsp
14     popq %rbp
15     ret
16 _start:
17     movq $4, A
18     movq $5, B
19     pushq B
20     pushq A
21     call soma
22     addq $16, %rsp
23     movq %rax, %rdi
24     movq $60, %rax
25     syscall

```

Algoritmo 3.6 – Tradução do programa do Algoritmo 3.5

- no retorno do fluxo ao chamador (linha 22), é que os parâmetros são desempilhados;
- dentro do procedimento soma, as referências a x e y são traduzidas respectivamente como 16(%rbp) e 24(%rbp);
- a variável local z está localizada em -8(%rbp).

3.2.4 Parâmetros passados por Referência e com Variáveis Locais

Os parâmetros da seção anterior foram passados por valor, ou seja, o valor foi copiado para a pilha. Isto faz com que a variável utilizada na chamada do procedimento não tenha o seu valor alterado quando do retorno.

Porém, por vezes é interessante que a variável que corresponde ao parâmetro efetivo alterado durante a execução do procedimento. Para satisfazer este tipo de necessidade, muitas linguagens de programação usam o conceito de passagem por referência. Na linguagem Pascal, um parâmetro formal passado por referência é precedido pela palavra reservada “var”.

A linguagem C não contém este tipo de construção, porém provê mecanismos para que o programador obtenha esta semântica colocando os símbolos & (e comercial) e * (asterisco) à frente das variáveis.

A ideia básica é que o valor a ser empilhado na chamada não é uma cópia do valor a ser passado, mas sim uma cópia do **endereço** da variável. Toda vez que a variável for acessada, deve ser indicado que se deseja acessar o valor *apontado* por aquele parâmetro e não o parâmetro em si. Como exemplo, considere o Algoritmo 3.7.

Neste algoritmo, o procedimento troca recebe dois parâmetros, x e y e troca os valores lá contidos. Ao final do programa apresentado no Algoritmo 3.7 principal, a terá o valor 2 e b terá o valor 1.

Alguns aspectos do programa C merecem destaque:

1. na chamada do procedimento troca (&a, &b) o símbolo & antes das variáveis indica que devem ser empilhados os *endereços* de a e b e não os valores lá contidos;
2. na assinatura do procedimento troca (void troca (long int* x, long int* y)), o asterisco antes dos parâmetros formais a e b indica que o procedimento troca assume que foram empilhados *endereços* no espaço destinado ao parâmetros formais x e y. Os dois parâmetros são *apontadores*, ou seja, contêm os endereços das variáveis a serem acessadas. Por esta razão, o comando z = *x; diz que o valor indicado pelo endereço contido em x deve ser copiado para z⁶.

⁶ Acredito que todo programador C já tenha cometido o erro de esquecer o asterisco - com resultados devastadores. Aliás, se existe um programador C que nunca cometeu este erro, então ele está fazendo algo errado!

```

1 long int a, b;
2 void troca (long int *x, long int *y)
3 {
4     long int z;
5     z = *x;
6     *x = *y;
7     *y = z;
8 }
9
10 int main (long int argc, char **argv)
11 {
12     a=1;
13     b=2;
14     troca (&a, &b);
15     return (0);
16 }

```

Algoritmo 3.7 – Programa com parâmetros passados por referência e com variáveis locais

C	<i>assembly</i>	Justificativa
<code>&a</code>	<code>\$A</code>	O símbolo \$ significa que o que vem a seguir deve ser tratado como uma constante. No caso, o rótulo de A.
<code>*x</code>	<code>movq 16(%rbp), %rax</code> <code>movq (%rax), %rbx</code>	A primeira linha copia o conteúdo de x para o registrador %rax, ou seja, um endereço. A segunda linha acessa o endereço apontado por %rax e o armazena em %rbx

Tabela 7 – Tradução de variáveis passadas por referência - Parte 1

Variável	endereço virtual	endereço léxico
A	0x60 011d	A
B	0x60 0125	B
x		16(%rbp)
y		24(%rbp)
z		-8(%rbp)

Tabela 8 – Endereços das variáveis do programa do Algoritmo 3.8

Entender estas construções em C não é suficiente ainda para traduzir o programa apresentado no Algoritmo 3.7 para *assembly*. É necessário também gerar instruções *assembly* que executam esta funcionalidade.

Dadas estas explicações, fica mais simples indicar a tradução do programa acima (Algoritmo 3.8).

O programa *assembly* reflete as ideias apresentadas anteriormente:

1. A chamada do procedimento (`troca (&a, &b)`) empilha os endereços das variáveis (linhas 26 e 27).
2. Dentro do procedimento, o acesso aos parâmetros formais é feito como indicado na Tabela 7. Para `*x`, veja linhas 10-11 e linhas 15-26. Para `*y` veja linhas 13-14 e linhas 18-19.

O funcionamento do mecanismo de passagem de parâmetros por referência é um tópico que muitos entendem como misterioso (ou simplesmente não entendem). No intuito de esclarecer este ponto, eu gerei o executável do programa contido no Algoritmo 3.8 e simulei a execução usando o `gdb` para obter os endereços das variáveis em tempo de execução. Os resultados obtidos estão na Tabela 8.

Em seguida, executei o programa passo a passo com o objetivo de explicitar a montagem do registro de ativação com valores reais dos registradores. O resultado pode ser visto na Figura 11.

- (I) Situação inicial no rótulo `_start`. Como A e B foram declaradas na seção `.data`, elas têm um endereço fixo.

```

1  .section .data
2      A: .quad 0
3      B: .quad 0
4  .section .text
5  .globl _start
6  troca:
7      pushq %rbp                # empilha %rbp
8      movq %rsp, %rbp          # faz %rbp apontar para novo R.A.
9      subq $8, %rsp            # long int z (em -8(%rbp))
10     movq 16(%rbp), %rax        # %rax := x
11     movq (%rax), %rbx         # %rbx := end. apont. por %rax (*x)
12     movq %rbx, -8(%rbp)       # z := *x
13     movq 24(%rbp), %rax       # %rax := y
14     movq (%rax), %rbx         # %rbx := end. apont. por %rax (*y)
15     movq 16(%rbp), %rax       # %rax := x
16     movq %rbx, (%rax)         # *x = *y
17     movq -8(%rbp), %rbx       # %rbx := z
18     movq 24(%rbp), %rax       # %rax := y
19     movq %rbx, (%rax)         # *y = z
20     addq $8, %rsp             # libera espaço alocado para z
21     popq %rbp                # restaura %rbp
22     ret
23 _start:
24     movq $1, A
25     movq $2, B
26     pushq $B                  # empilha endereço de B
27     pushq $A                  # empilha endereço de A
28     call troca
29     addq $16, %rsp             # libera espaço dos parametros
30     movq $0, %rdi
31     movq $60, %rax
32     syscall

```

Algoritmo 3.8 – Tradução do Algoritmo 3.7

(2) Situação após a montagem do registro de ativação após alocar a variável local (linha 9). Destaco:

1. o conteúdo no espaço reservado para a primeira e segunda variáveis locais (“emprestei” o nome delas do programa em C). A instrução `pushq $B` empilhou o *endereço* de B, ou seja, 0x60 0125. Este é o valor da variável em 24(%rbp) (y). Há uma seta saindo deste local e apontando para B, o que dá sentido ao termo *apontador*. No programa C teríamos aqui `y=0x60 0125` e `*y=2`.
2. a instrução `call` empilhou o endereço de retorno (0x40 0109). O procedimento chamado salvou o antigo %rbp (0x0) e também alocou espaço para a variável local.

(3) Corresponde a execução das instruções 10 até 12 (linha 5 do Algoritmo 3.7) que copia o conteúdo do endereço apontado por 16(%rbp) em -8(%rbp).

(4) Corresponde a execução das instruções 13 até 16 (linha 6 do Algoritmo 3.7) que copia o conteúdo do endereço apontado por 16(%rbp) no endereço apontado por 24(%rbp).

(5) Corresponde a execução das instruções 17 até 19 (linha 7 do Algoritmo 3.7) que copia o conteúdo da variável local para o endereço apontado por 24(%rbp).

(6) A execução prossegue liberando a variável local (linha 20), restaurando %rbp (linha 21) retornando do procedimento (linha 22) e finalmente desempilhando os parâmetros (linha 29) quando a pilha assume esta configuração.

3.2.5 Chamadas Recursivas

Para finalizar o estudo sobre o funcionamento de chamadas de procedimento, esta seção apresenta um exemplo de um programa recursivo. A ideia é fixar todos os conceitos apresentados sobre chamadas de procedimento, em especial a forma com que os registros de ativação são empilhados e desempilhados para um mesmo procedimento.

O Algoritmo 3.9 contém todos estes elementos. Ele calcula o fatorial de um número (no caso, de 3). Existem maneiras muito mais elegantes e muito mais eficientes para implementar um programa que calcula o fatorial, porém este

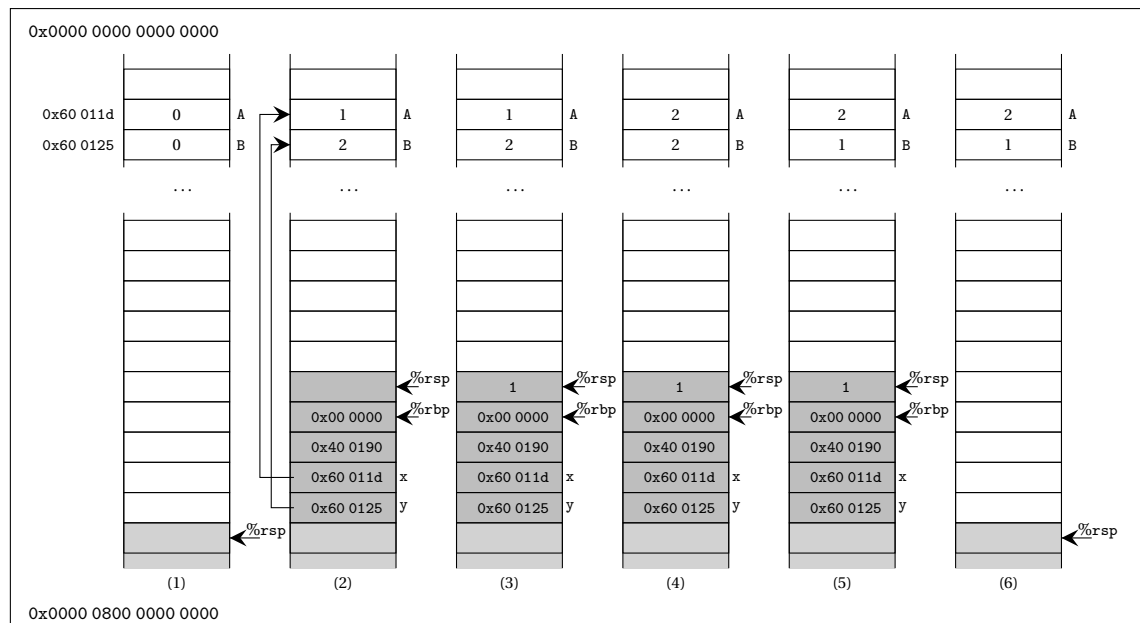


Figura 11 – Passagem de parâmetros por referência

```

1 void fat (long int *res, long int n)
2 {
3     long int r;
4     if (n<=1)
5         *res=1;
6     else {
7         fat (res, n-1);
8         r = *res * n;
9         *res=r;
10    }
11 }
12
13 int main (long int argc, char **argv)
14 {
15     long int x;
16     fat (&x, 3);
17     return (x);
18 }

```

Algoritmo 3.9 – Programa Recursivo

programa foi implementado desta maneira (até certo ponto confusa) para incluir parâmetros passados por referência, parâmetros passados por valor e variáveis locais (observe que a variável “r” é desnecessária).

O Algoritmo 3.9 também contém um dos elementos que mais confunde os programadores iniciantes da linguagem C: o uso (ou não) dos caracteres modificadores `&` e `*` à frente de uma variável nas passagens de parâmetro por referência.

O primeiro parâmetro de `fat` é uma variável passada por referência, ou seja, o seu conteúdo é um endereço. O procedimento é chamado duas vezes. Na chamada de procedimento da linha 15, o parâmetro real `x` é precedido por um `&`, ou seja, é empilhado o endereço de `x`. Dentro do do procedimento `fat` é possível acessar o endereço de `x` com `*res` (ou seja, `res` é um apontador para `x`).

Na segunda chamada de procedimento (linha 7), o primeiro parâmetro é `res` e não `*res` ou `&res`. Este programa está correto da forma com que foi apresentado, e ficará mais claro na simulação da execução a seguir.

A tradução do Algoritmo 3.9 é apresentada no Algoritmo 3.10 (veja Exercício 3.7).

A novidade nesta tradução é como empilhar o parâmetro `x` (variável local a `main`) declarado na linha 15), que não foi apresentada na Tabela 7. Para isso, a Tabela 9 mostra novamente a Tabela 7 acrescentando a construção `&x`, onde `x` é

```

1  .section .data
2  .section .text
3  .globl _start
4  fat:
5      pushq %rbp                # empilha %rbp
6      movq %rsp, %rbp          # faz %rbp apontar para novo R.A.
7      subq $8, %rsp            # long int r (em -8(%rbp))
8      movq 24(%rbp), %rax       # %rax := n
9      movq $1, %rbx            # %rbx := 1
10     cmpq %rbx, %rax
11     jg else                   # desvia se n>1
12     movq 16(%rbp), %rax       # %rax := res
13     movq $1, (%rax)           # *res := 1
14     jmp fim_if
15 else:
16     movq 24(%rbp), %rax       # %rax := n
17     subq $1, %rax             # %rax := n-1
18     pushq %rax                # empilha %rax
19     pushq 16(%rbp)            # empilha res
20     call fat
21     addq $16, %rsp            # desempilha parametros
22     movq 16(%rbp), %rax       # %rax := res
23     movq (%rax), %rax          # %rax := *res
24     movq 24(%rbp), %rbx       # %rbx := n
25     imul %rbx, %rax           # %rax := %rax * %rbx
26     movq %rax, -8(%rbp)       # r := *res * n
27     movq -8(%rbp), %rax       # %rax := r
28     movq 16(%rbp), %rbx       # %rbx := res
29     movq %rax, (%rbx)         # *res = r
30     fim_if:
31     addq $8, %rsp             # libera espaco alocado para r
32     popq %rbp                # restaura %rbp
33     ret
34 _start:
35     pushq %rbp                # empilha %rbp
36     movq %rsp, %rbp          # faz %rbp apontar para novo R.A.
37     subq $8, %rsp            # long int x (em -8(%rbp))
38     pushq $3                  # empilha constante 3
39     movq %rbp, %rax           # %rax := %rbp
40     subq $8, %rax             # %rax := %rbp-8 (endereço de x)
41     pushq %rax                # empilha endereço de x
42     call fat
43     addq $16, %rsp            # libera espaco dos parametros
44     movq -8(%rbp), %rdi
45     movq $60, %rax
46     syscall

```

Algoritmo 3.10 – Tradução do Algoritmo 3.9

uma variável local.

A execução do programa *assembly* do Algoritmo 3.10 implica na criação de quatro registros de ativação. O primeiro para *main* e os seguintes para $n = 3$, $n = 2$ e $n = 1$.

A Figura 12 mostra a situação da pilha após cada chamada (ou seja, após montar cada registro de ativação). Nesta Figura foram omitidos os parâmetros de *main*.

(1) O registro de ativação de *main* após alocar espaço para a variável local *x*.

(2) A primeira chamada de *fat* com $n = 3$:

- à esquerda estão indicados os nomes dos parâmetros e variáveis locais (*n*, *res* e *r*) já com o seu conteúdo: $n = 3$ e $res \rightarrow x$ (*res* contém o endereço de *x*, logo *res* “aponta” para *x*).
- O valor de *%rbp* em (2) é o endereço contido em *%rbp* no registro de ativação *main* em (1).

C	assembly	Justificativa
&a	\$A	O símbolo \$ significa que o que vem a seguir deve ser tratado como uma constante. No caso, o rótulo de A.
*x	movq 16(%rbp), %rax movq (%rax), %rbx	A primeira linha copia o conteúdo de x para o registrador %rax, ou seja, um endereço. A segunda linha acessa o endereço apontado por %rax e o armazena em %rbx
&x	movq %rbp, %rax subq 8, %rax	Quando x é uma variável localizada em -8(%rbp), ela está localizada no endereço %rbp - 8

Tabela 9 – Tradução de variáveis passadas por referência - Parte 2. Aqui, "a" é uma variável global e "x" é uma variável local

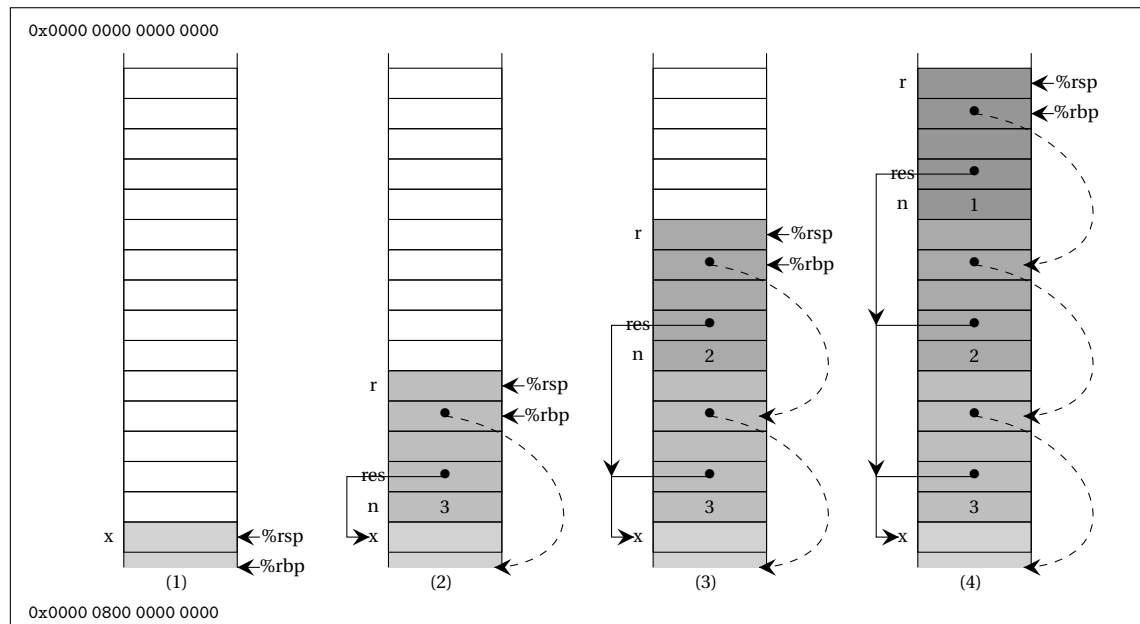


Figura 12 – Registros de ativação da execução do programa do Algoritmo 3.10

(3) Registro de ativação fat com $n = 2$:

- Assim como em (2), estão indicados à esquerda os nomes dos parâmetros e variáveis locais do novo registro de ativação. Observe que eles se sobrepõem aos nomes do registro de ativação anterior. Assim, $n = 2$. O parâmetro *res* contém o mesmo valor de *res* do parâmetro anterior, ou seja, $res \rightarrow x$ (veja Exercício 3.6).
- O endereço contido em %rbp aponta para o registro de ativação anterior, (2), mais especificamente para o local onde estava armazenado %rbp naquele registro de ativação. Este, por sua vez aponta para o %rbp do registro de ativação de *main* em (1). Isto mostra uma lista ligada de %rbps.

(4) Registro de ativação fat com $n = 1$, o último.

- Mais uma vez foram indicados os parâmetros e variáveis locais à esquerda e os nomes se sobrepõem aos nomes do registro de ativação anterior. Novamente, $res \rightarrow x$

Uma das informações mais importantes da Figura 12 é a forma que é usado o registrador %rbp:

- Como já era conhecido, o registrador %rbp aponta sempre para um local fixo na moldura do registro de ativação (veja Figura 10). Porém, o que a Figura 12 mostra claramente é que ao mudar o valor de %rbp, tem-se um novo registro de ativação em um local diferente da memória e que as informações armazenadas neste registro de ativação independem dos demais registros de ativação. Para comprovar esta afirmação observe que o endereço léxico -8(%rbp) atinge um endereço de memória diferente em cada novo registro de ativação. Isto pode ser visto

Parâmetro	Registrador
1	%rdi
2	%rsi
3	%rdx
4	%rcx
5	%8
6	%9
7, 8, 9, ...	Pilha

Tabela 10 – Convenção de associação de parâmetros e registradores

na Figura 12: a execução da instrução `movq $1, -8(%rbp)` coloca a constante 1 em endereços reais diferentes, mas sempre no registro de ativação que estiver no topo sem afetar os demais registros de ativação.

- Dada a importância do registrador `%rbp`, seus valores anteriores (os que correspondem aos registros de ativação anteriores) estão armazenados numa lista ligada onde o cabeça é o endereço apontado pelo registrador `%rbp` e o último elemento é o primeiro valor de `%rbp` em `main`.
- Isto explica porque na linguagem C, as únicas variáveis que podem ser acessadas pelo nome em um procedimento são aquelas declaradas naquele procedimento e as variáveis globais.

3.3 Aspectos Práticos

O modelo de registros de ativação apresentado até aqui é semelhante ao adotado no linux para processadores da família x86, a precursora de 32 bits do AMD64.

Como o AMD64 apresenta uma série de diferenças com relação aos x86, em especial um maior número de registradores, a ABI-AMD64[51] incluiu algumas mudanças. A Seção 3.3.1 apresenta como o mecanismo de passagem de parâmetros foi adaptado para usar registradores ao invés da pilha enquanto a Seção 3.3.2 descreve como proceder para salvar os valores de variáveis entre chamadas de procedimento. A Seção 3.3.3 mostra como utilizar estas alterações para escrever programas que incluem bibliotecas já existentes, exemplificando com a biblioteca `libc` para permitir impressão e leitura com as funções `printf` e `scanf`. A Seção 3.3.4 mostra como acessar os parâmetros `argc`, `argv` e `arge` do procedimento `main` da linguagem C. A Seção 3.3.5 apresenta uma área disponível acima do registrador `%rsp` chamada *red zone* e a Seção 3.3.6 apresenta alguns pontos de potencial vulnerabilidade do uso da pilha.

Por fim, a Seção 3.4 descreve como implantar registros de ativação no MIPS.

3.3.1 Parâmetros passados em registradores

Como sugerido pela Figura 34 (página 177) o tempo para acessar informações armazenadas na memória é maior do que o tempo para acessar informações em registradores na CPU. Sendo assim, programas que maximizam a quantidade de variáveis mapeadas em registradores apresentam um desempenho melhor do que programas equivalentes que não o fazem.

Esta preocupação com o desempenho foi incluída na ABI-64 com uma convenção para utilizar registradores como local de armazenamento de parâmetros de chamadas de procedimento.

A convenção está descrita em [51] e está indicada na Tabela 10. Como ela é adotada por todas as bibliotecas da linguagem C do linux, é possível integrar os programas escritos em *assembly* a estas bibliotecas seguindo esta convenção⁷.

A tabela mostra que os primeiros seis parâmetros são colocados em registradores. A partir do sétimo parâmetro, são colocados na pilha.

3.3.2 Como salvar variáveis em chamadas de procedimento

Conforme descrito na Seção 3.3.1, tanto parâmetros quanto variáveis locais devem ser alocados preferencialmente em registradores para minimizar o acesso à memória e consequentemente melhorar o desempenho dos programas.

⁷ A convenção proposta também explica onde passar parâmetros reais (ponto flutuante) e outros, mas isto não será abordado neste livro

Isto significa que no programa apresentado no Algoritmo 3.11 (reimpressão do Algoritmo 3.9), as variáveis do procedimento `fat` deveriam ser alocadas em registradores. Pela Tabela 10, a variável `res` deve ser alocada ao registrador `%rdi` e a variável `n` no registrador `%rsi`. Já para a variável `r` deve ser escolhido outro registrador. Como `%rax` e `%rbx` são usados como intermediários, vamos considerar que seja mapeado no registrador `%rcx`.

```

1 void fat (long int *res, long int n)
2 {
3     long int r;
4     if (n<=1)
5         *res=1;
6     else {
7         fat (res, n-1);
8         r = *res * n;
9         *res=r;
10    }
11 }
12
13 int main (long int argc, char **argv)
14 {
15     long int x;
16     fat (&x, 3);
17     return (x);
18 }

```

Algoritmo 3.11 – Reimpressão do programa recursivo do Algoritmo 3.9

A questão interessante é o que acontece a partir da segunda chamada recursiva. Como o código é estático, o mesmo mapeamento é usado em cada novo registro de ativação, **sobrescrevendo** o valor contido nos mesmos registradores usados para armazenar as variáveis do primeiro registro de ativação.

Na “ida” das chamadas recursivas, tudo funciona. O problema é que na volta, os valores anteriores estarão perdidos.

Este problema não ocorre só na recursão, mas em qualquer chamada de procedimento. Seja R_1 o conjunto de registradores usado em um procedimento p_1 e seja R_2 o conjunto de registradores usado em um procedimento p_2 . Considere o que ocorre quando p_1 chama p_2 . Se $R_1 \cap R_2 \neq \emptyset$, então existe pelo menos uma variável em p_2 que, se alterada, alterará o valor de uma variável de p_1 .

Por exemplo, considere o Algoritmo 3.12. Suponha que a variável `i` é mapeada no registrador `%rdi`, e suponha que a sub-rotina `printf` também usa o registrador `%rdi` (ou seja, que $R_1 \cap R_2 = \{\%rdi\}$), alterando valor de `%rdi` para qualquer valor maior que 101. Isto significa que a sub-rotina `printf` só será executada uma das 100 vezes pretendida.

```

1 ...
2 for (i=0; i<100; i++) {
3     printf ("%d\n", i);
4 }
5 ...

```

Algoritmo 3.12 – Programa com um `printf` dentro de um `for`

Seja R_1 a tabela contendo a lista dos registradores e os valores contidos em cada um destes registradores antes da chamada de um procedimento, digamos antes de chamar o procedimento `printf`. Seja R_2 uma tabela contendo a lista dos registradores e os valores contidos em cada um deles no retorno do procedimento `printf`.

Um programador otimista pode pensar que, após o `printf`, $R_1 = R_2$, porém raramente isto será verdadeiro, uma vez que o procedimento `printf` poderá utilizar alguns destes registradores, sobrescrevendo-os. Por esta razão, o normal é que $R_1 \neq R_2$.

Por exemplo, considere que `%rcx=0x10` antes da instrução `call printf`. Quando do retorno do procedimento (ou seja, na linha seguinte à instrução `call`), podemos ter `%rcx=0xff`, ou qualquer outro valor, inclusive `%rcx=0x10`.

Se `%rcx` fosse usado para armazenar uma variável local antes da chamada a `printf`, então esta variável misteriosamente iria mudar de valor ao retornar (bem, já não é tão misterioso agora...).

Para contornar este problema, a solução é salvar os valores contidos em registradores em algum lugar, mais especificamente na pilha. Para tal existem duas abordagens:

salvar registradores no procedimento chamador (*caller save*) Aqui, o conjunto de registradores utilizados pelo chamador são salvos antes de empilhar os parâmetros da chamada do procedimento e restaurados na volta;

salvar registradores no procedimento chamado (*callee save*) Aqui, o conjunto de registradores que serão usados no procedimento são empilhados após alocar as variáveis locais (se houver) e desempilhados antes de liberar as variáveis locais.

Estas duas abordagens podem ser facilmente implementadas com o uso das instruções `push` e `pop`. Porém, a convenção adotada usa um método diferente: assim que finaliza a construção de um registro de ativação (após alocar as variáveis locais), o procedimento chamado abre um espaço na pilha para abrigar os dois conjuntos de registradores.

O Algoritmo 3.13 descreve como salvar os registradores pelo *caller*, ou seja, salvar os registradores que o procedimento chamador quer manter inalterados pela chamada da linha 9 (`call procedimento`).

Considere que este procedimento quer salvar os parâmetros passados para ele (em `%rsi`, `%rdi` e `%rdx`). O primeiro passo é abrir espaço para salvá-los na pilha como indicado na linha 2 do Algoritmo 3.13. Antes de empilhar os parâmetros do procedimento deve salvar os registradores na pilha (linhas 5-7). Após retornar do procedimento, deve restaurar os registradores (linhas 11-13). Ao final do procedimento, é necessário liberar o espaço alocado para os salvamentos (linha 15).

O Algoritmo 3.14 descreve como salvar os registradores pelo *callee*, ou seja, salvar os registradores que serão alterados naquele procedimento chamado. Suponha que o procedimento `funcao` irá alterar os valores contidos nos registradores `%rbx`, `%r12` e `%r13`. Para que o valor destes registradores na entrada do procedimento sejam iguais a valor na saída do procedimento, ele salva os valores na entrada (linhas 8-10) e os restaura na saída (linhas 12-14).

```

1  ...
2  subq  $24, %rsp                # Locais para salvar %rsi, %rdi e %rdx
3  ...
4  ...                            # salvamento ANTES de chamar procedimento
5  movq  %rsi, -8(%rbp)           # salva %rsi
6  movq  %rdi, -16(%rbp)          # salva %rdi
7  movq  %rdx, -24(%rbp)          # salva %rdx
8  <empilha Parametros>
9  call  procedimento
10 <desempilha Parametros>
11 movq  -8(%rbp), %rsi           # restaura %rsi
12 movq  -16(%rbp), %rdi          # restaura %rdi
13 movq  -24(%rbp), %rdx          # restaura %rdx
14 ...
15 addq  $24, %rsp                # libera espaco alocado
16 ...

```

Algoritmo 3.13 – Abordagem de salvar registradores pelo *caller*

Cada uma das duas abordagens têm contexto diferente: enquanto o chamador defende-se dos atos do procedimento chamado, o procedimento chamado atua preventivamente. Porém, isto pode fazer com que um registrador seja salvo por ambos.

Para evitar este problema, a ABI-AMD64 [51] indica quais registradores são preservados entre chamadas de procedimento (ou seja que devem ser salvos pelo *callee* se usados). A Tabela 11 apresenta esta convenção.

A tabela mostra que o procedimento chamado, o *callee*, é responsável por garantir que os valores contidos nos registradores `%rbx`, `%r12`-`%r15` contenham, na saída do procedimento, os mesmos valores que continham na entrada (veja modelo no Algoritmo 3.14).

A tabela só mostra os registradores que devem ser preservados pelo *callee*, e a partir disto, é possível deduzir que o procedimento chamador, o *caller*, é o responsável por preservar os registradores `%rax`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%r8`, `%r9`, `%r10` e `%r11` (veja modelo no Algoritmo 3.13). Isto, claro, só se os utilizar de alguma forma.

Os registradores `%rsp` e `%rbp` são preservados no registro de ativação do procedimento chamado.

3.3.3 Uso de Bibliotecas

Até aqui, todos os programas apresentados colocam o resultado da execução na variável de ambiente `$_`, uma variável de oito bits que não é apropriada para armazenar qualquer inteiro maior que 255 (tente executar o programa do Algoritmo 3.10 com `n = 6` e veja o resultado).

```

1 funcao:
2     pushq %rbp
3     movq %rsp, %rbp
4     ...
5     subq $24, %rsp                # Locais para salvar %rbx, %r12 e %r13
6     ...                          # salvamento NA ENTRADA do procedimento
7     movq %rbx, -8(%rbp)          # salva %rbx
8     movq %r12, -16(%rbp)         # salva %r12
9     movq %r13, -24(%rbp)         # salva %r13
10    ...                          # usa %rbx, %r12, %r13
11    movq -8(%rbp), %rbx          # restaura %rbx
12    movq -16(%rbp), %r12         # restaura %r12
13    movq -24(%rbp), %r13         # restaura %r13
14    addq $24, %rsp              # libera espaco alocado
15    ...
16    popq %rbp
17    ret
18

```

Algoritmo 3.14 – Abordagem de salvar registradores pelo *callee*

Registrador	Uso	Preservado
%rax	Temporário, 1º retorno procedimento	Não
%rbx	Temporário	Sim
%rcx	4º parâmetro inteiro	Não
%rdx	3º parâmetro inteiro	Não
%rsp	<i>stack-pointer</i>	Sim
%rbp	<i>base-pointer</i>	Sim
%rsi	2º parâmetro inteiro	Não
%rdi	1º parâmetro inteiro	Não
%r8	5º parâmetro inteiro	Não
%r9	6º parâmetro inteiro	Não
%r10	Temporário	Não
%r11	Temporário	Não
%r12-r15	Temporário	Sim

Tabela 11 – Uso de registradores de propósito geral (adaptado de [51])

A convenção de passagem de parâmetros em registradores apresentada na Seção 3.3.1 é adotada pelas bibliotecas da linguagem C. Isto significa que é possível integrar programas *assembly* a procedimentos já existentes nestas bibliotecas, como por exemplo, a `libc`.

A `libc` é uma biblioteca que contém milhares de funções de uso geral que incluem gerenciamento de memória, entrada e saída, manipulação de *strings* entre outras. Atualmente o linux adota a implementação da versão da `libc` da FSF⁸ chamada `glibc`, mas que no linux é chamada somente `libc`.

Como todos os programas básicos do linux, a `libc` é um código aberto, que pode ser encontrado facilmente na internet⁹. Porém, existe mais de uma implementação da `libc` disponível. A curiosa história do número da versão da `libc` (assim como a implementação) adotada no linux é narrada por Rick Moen¹⁰.

Apesar de todas as funções da `libc` estarem armazenadas em um único arquivo, as assinaturas destas funções são encontradas em diversos arquivos de cabeçalho, como por exemplo: `stdio.h`, `malloc.h`, `string.h`, entre muitas outras.

Esta seção descreve como usar duas funções implementadas pela `libc`: `printf` e `scanf`. Para utilizar estas duas funções, são necessários alguns cuidados:

1. Dentro do programa *assembly*, os parâmetros devem ser colocados nos registradores conforme a Tabela 10.

⁸ Free Software Foundation

⁹ <http://www.gnu.org/software/libc/libc.html>

¹⁰ http://linuxmafia.com/faq/Licensing_and_Law/forking.html

2. Seguir rigorosamente a assinatura de `printf` e `scanf`, pois ao contrário da compilação de um programa na linguagem C, o programa *assembly* não irá avisar potenciais erros.
3. Ao ligar o programa, é necessário incluir a própria `libc`. Como faremos a ligação dinâmica, também é necessário incluir a biblioteca que contém o ligador dinâmico.

Para exemplificar o processo, considere o Algoritmo 3.15, e sua tradução, o Programa 3.16.

```

1  #include <stdio.h>
2  int main (long int argc, char **argv)
3  {
4      long int x, y;
5      printf("Digite dois numeros:\n");
6      scanf ("%ld %ld" , &x, &y);
7      printf("Os numeros digitados foram %ld %ld \n" , x, y);
8
9      return (0);
10 }
```

Algoritmo 3.15 – Uso de `printf` e `scanf`

```

1  .section .data
2      str1: .string "Digite dois numeros:\n"
3      str2: .string "%d %d"
4      str3: .string "Os numeros digitados foram %d %d\n"
5  .section .text
6  .globl main
7  main:
8      pushq %rbp
9      movq %rsp, %rbp
10     subq $16, %rsp
11     mov $str1, %rdi
12     call printf
13     movq %rbp, %rax
14     subq $16, %rax
15     movq %rax, %rdx
16     movq %rbp, %rax
17     subq $8, %rax
18     movq %rax, %rsi
19     mov $str2, %rdi
20     call scanf
21     movq -16(%rbp), %rdx
22     movq -8(%rbp), %rsi
23     mov $str3, %rdi
24     call printf
25     movq $60, %rax
26     syscall
```

Algoritmo 3.16 – Tradução do Algoritmo 3.15

O programa apresentado no Algoritmo 3.16 tem novidades.

- A entrada é no rótulo `main`. O rótulo `_start` será incluído na linha de ligação.
- A chamada das funções `printf` (linhas 12 e 24) e `scanf` (linha 20) ocorrem como se estas funções estivessem presentes neste arquivo.
- Os parâmetros são colocados em registradores conforme indicado na Tabela 10 (linhas 11, 13-15, 16-18, 19, 21-23).
- Obviamente não é necessário desempilhar nenhum parâmetro.

Para gerar o executável, o processo é mais complicado, pois o programa executável precisa ser compatível com o programa C além de incluir a biblioteca `libc` (`-lc`).

Estamos fazendo ligação dinâmica, por isso a opção `-dynamic-linker` que indica qual ligador usar. O compilador `gcc` automaticamente inclui os arquivos `crt1.o`, `crti.o` e `crtn.o`, mas quando se usa o programa `ld`, é necessário incluí-los explicitamente¹¹.

```
> as printfScanf.s -o printfScanf.o -g
> ld printfScanf.o -o printfScanf -dynamic-linker /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2 \
/usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o \
/usr/lib/x86_64-linux-gnu/crtn.o -lc
> ./printfScanf
Digite dois números:
1 2
Os números digitados foram 1 2
>
```

As funções `printf` e `scanf` não estão implementadas aqui, e sim na biblioteca `libc`, disponível em `/usr/lib/libc.a` (versão estática) e `/usr/lib/libc.so` (versão dinâmica). Neste exemplo utilizamos a biblioteca dinâmica para gerar o programa executável.

A opção `-dynamic-linker /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2` indica qual o ligador dinâmico que será o responsável por carregar a biblioteca `libc`, indicada em `-lc`, em tempo de execução.

Uma curiosidade: é possível inserir instruções *assembly* em um código C usando o procedimento `__asm()`. Veja exemplo no Algoritmo 3.17. Observe que a variável `x` foi declarada como `long int` que ocupa 8 bytes.

```
1 #include <stdio.h>
2 int main (long int argc, char **argv)
3 {
4     long int x;
5     __asm__ ("movq $100, -8(%rbp)");
6     printf("x=%ld\n", x);
7
8     return (0);
9 }
```

Algoritmo 3.17 – Uso de instruções *assembly* em C

3.3.4 O procedimento main da linguagem C

Cada linguagem de programação define um local para iniciar o programa. Na linguagem C, este local é o procedimento `main`. Até este momento, associamos este procedimento ao rótulo `_start`, que indica o local onde um programa *assembly* deve iniciar.

Esta seção explica como dar a funcionalidade correta para o procedimento `main` da linguagem C, dando especial importância aos seus parâmetros formais: um inteiro (`argc`) e dois vetores de endereços chamados `argv` e `arge`.

Os parâmetros `argc` e `argv` estão relacionados aos argumentos passados na linha de comando enquanto que `arge` está relacionado com as variáveis de ambiente.

É mais simples explicar estes parâmetros e sua funcionalidade através de um exemplo. Considere o Algoritmo 3.18.

Ele imprime todos os parâmetros formais de `main`, um por linha. Cada linha contém o endereço onde está localizado o parâmetro formal ("%p" imprime em hexadecimal), o nome do parâmetro formal com índice e o *string* correspondente.

Após compilado, eu executei o programa com os argumentos "111 222 333 444" em linha de comando e obtive o seguinte resultado:

```
0x7ffc67961398]: argc=4
[0x7ffc67961498]: argv[0]=./imprimeArgs 0x7ffc67962146
[0x7ffc679614a0]: argv[1]=111 0x7ffc67962154
[0x7ffc679614a8]: argv[2]=222 0x7ffc67962158
```

¹¹ para mais informações sobre estes arquivos, consulte a Seção 9.2.8

```

1 #include <stdio.h>
2 int main (long int argc, char **argv, char **arge)
3 {
4     long int i;
5
6     // ----- Imprime argc -----
7     printf("[%p]: argc=%ld\n", &argc, argc);
8
9     // ----- Imprime argv -----
10    // for (i=0; argv[i] != NULL; i++) : Tambem funciona
11    for (i=0; i<argc; i++)
12        printf("[%p]: argv[%ld]=%s %p\n", &argv[i], i, argv[i], argv[i]);
13    printf("[%p]: argv[%ld]=%s %p\n", &argv[i], i, argv[i], argv[i]);
14
15    // ----- Imprime arge -----
16    for (i=0; arge[i] != NULL; i++)
17        printf("[%p]: arge[%ld]=%s %p\n", &arge[i], i, arge[i], argv[i]);
18    printf("[%p]: arge[%ld]=%s %p\n", &arge[i], i, arge[i], argv[i]);
19
20    return (0);
21 }

```

Algoritmo 3.18 – Programa que imprime os argumentos e variáveis de ambiente

```

[0x7ffc679614b0]: argv[3]=333 0x7ffc6796215c
[0x7ffc679614b8]: argv[4]=(null) (nil)
[0x7ffc679614c0]: arge[0]=_system_type=Linux 0x7ffc67962146
...
[0x7ffc67961580]: arge[24]=USER=bmuller 0x7ffc67962377
...
[0x7ffc679615c0]: arge[32]=HOME=/home/bmuller 0x7ffc6796246e
...
[0x7ffc67961778]: arge[87]=(null) 0x7ffc67962dee

```

São vários os destaques:

- `argc` é usado para indicar quantos parâmetros formais `argv` estão presentes. Por isso, o laço da linha 11 ocorre até este limite;
- o primeiro argumento é o nome do programa (`argv[0]=./imprimeArgs`);
- para listar as variáveis de ambiente, o laço é diferente (linha 16). Ele persiste até encontrar uma entrada de `arge[i]` cujo conteúdo é `NULL` (que é listada como `arge[87]`) na saída da execução;
- a quantidade de variáveis de ambiente é grande. No meu caso, foram listadas 86 variáveis e eu destaquei só duas das mais conhecidas;
- a linha 10 apresenta uma forma alternativa para percorrer o vetor `argv`. Ela não usa o parâmetro `argc`, e sim o conhecimento que, assim como o vetor `arge`, seu último elemento é sucedido por um elemento cujo conteúdo é `NULL`. Veja a impressão de `argv[4]`;
- a primeira coluna indica o endereço daquela variável. Por exemplo, o endereço de `argv[0]` é `[0x7ffc67961498]`;
- a segunda coluna indica o nome da variável;
- a terceira coluna indica o valor da variável. Por exemplo, `argc[1]=111\0`;
- a última coluna indica o valor contido em cada variável. Por exemplo, `argv[1]=0x7ffc67962154`, o endereço de memória onde inicia o `string "111\0"`;
- observe que o valor contido em cada variável (quarta coluna) somado ao tamanho do `string` apontado corresponde ao valor contido na variável seguinte. Por exemplo, `./imprimeArgs\0` ocupa 14 bytes, logo `argv[0] + 14 = argv[1]`, ou seja, `0x7ffc67962146 + 0xE = 0x7ffc67962154`.

Esta explicação gera novas dúvidas¹², se `argv[1]` contém um endereço, onde é que fica armazenado o `string` correspondente. Quem o colocou lá, entre (várias) outras dúvidas.

¹² “A dúvida cresce com conhecimento” (Goethe)

Vamos tentar explicar isto com mais detalhes. Quando um programa é colocado em execução, uma parte do sistema operacional, chamada carregador (*loader*) executa diversas tarefas para disponibilizar o ambiente de execução, dentre as quais, alocar o espaço virtual de endereçamento. Em um local específico deste espaço, ele copia cada um dos argumentos passados na linha de comando e acrescenta um `"\0"` ao final de cada um. Na linguagem C o símbolo `"\0"` indica o fim do *string*. Na prática, este símbolo é representado por um byte com todos os bits desligados, ou seja, um byte igual a zero, também representado por `NULL`.

O resultado desta ação pode ser vista na Figura 13 ao lado, que descreve a localização dos parâmetros e conteúdo `argc` e `argv`. A Figura mostra os três parâmetros quando se executa um programa com a linha de comando abaixo:

```
> ./imp 111 222 333
```

Como são quatro parâmetros, `argc = 4`. O parâmetro `argv` contém o endereço de início de um vetor de ponteiros, que são `argv[0]`, `argv[1]`, `argv[2]` e `argv[3]`. `argv[0]` contém o endereço de início do *string* "imp\0", `argv[1]` contém o endereço de início do *string* "111\0", `argv[2]` contém o endereço de início do *string* "222\0" e `argv[3]` contém o endereço de início do *string* "333\0".

Observe que `argv[4]=NULL`, indicando fim do vetor, o que explica porque o comando `for` da linha 10 do Algoritmo 3.18 também pode ser usado.

O caso de `arge` não é detalhado na Figura 13, mas a situação é semelhante. O carregador armazena em algum lugar da pilha todas as variáveis de ambiente em sequência. Em seguida, preenche `arge[0]`, `arge[1]`, `arge[2]`, etc. com os endereços de início de um *string* em um formato "NOME_DA_VARIÁVEL_DE_AMBIENTE=valor\0", como por exemplo, "PWD=/home/bmuller\0", "LC_PAPER=pt_BR.UTF-8\0", etc.

Vale a pena destacar que o primeiro elemento do vetor `arge`, `arge[0]`, começa logo após o último `argv`, no caso, após `argv[4]=NULL`.

Em linux, isto se aplica a todos os programas executáveis, independente da linguagem em que foram escritos.

Agora vamos analisar como acessar estes parâmetros em um programa *assembly*. A primeira coisa a observar é que são três parâmetros, e que pelo mapeamento da Tabela 10, teremos:

Parâmetro	Registrador
<code>argc</code>	<code>%rdi</code>
<code>argv</code>	<code>%rsi</code>
<code>arge</code>	<code>%rdx</code>

A próxima observação é que na assinatura do procedimento `main`, temos `argv` e `arge` precedidos de dois asteriscos. Esta notação indica que a variável contém o endereço de início de um vetor de endereços.

Com isto explicado, podemos indicar como acessar os *strings* apontados por `argv[0]`, `argv[1]`, etc. Nos exemplos abaixo, o endereço do *string* é colocado no registrador `%rax`.

Observe que o endereço de `argv[i]` é obtido pela fórmula

$$\&argv[i] = (\&argv + i * 8)$$

onde a constante 8 indica o número de bytes usados por uma variável do tipo endereço.

Agora que já sabemos acessar os parâmetros do procedimento `main`, apresentamos no Algoritmo 3.19 um programa *assembly* que imprime o `argv[0]`.

As linhas 13 e 14 colocam em `%rsi` o início do *string* indicado por `argv[0]` conforme descrito na Tabela 12.

3.3.5 Red Zone

Em princípio, a área acima de `%rsp` é uma "área minada": o acesso é inválido e a tentativa de acessar estes endereços ocasiona o cancelamento do programa pelo sistema operacional.

Isto nem sempre ocorre, pois para efeito de melhoria de desempenho, o teste é feito analisando se a página é inválida, e não os endereços específicos.

A ABI-AMD64 [51] especifica a existência de uma área que não é "minada". Esta área vai do endereço (`%rsp`) até `-128(%rsp)` conforme especificado na ABI:

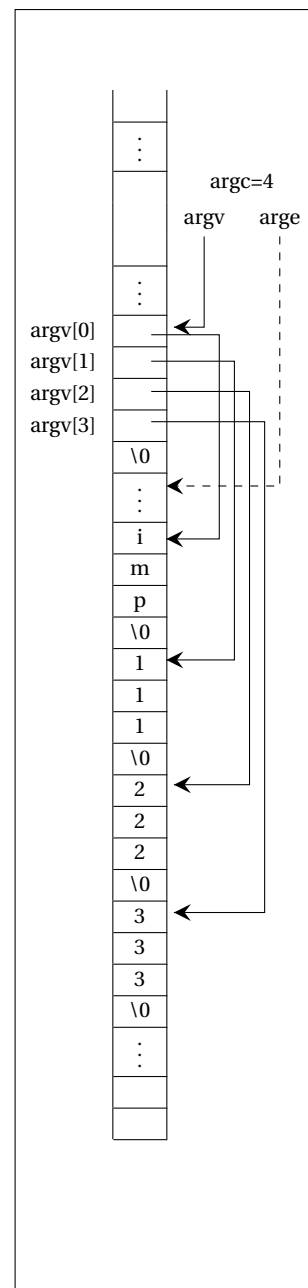


Figura 13 – Apontadores `argc`, `argv` e `arge`

Parâmetro	Acesso
argv[0]	movq %rsi, %rax movq (%rax), %rax
argv[1]	movq %rsi, %rax addq \$8, %rax movq (%rax), %rax
argv[i]	movq %rsi, %rax movq I, %rbx mulq \$8, %rbx addq %rbx, %rax movq (%rax), %rax
arge[0]	movq %rdx, %rax movq (%rax), %rax
arge[1]	movq %rdx, %rax addq \$8, %rax movq (%rax), %rax

Tabela 12 – Instruções para acessar os parâmetros em main

```

1  .section .data
2      str1: .string "Meu nome eh %s\n"
3  .section .text
4  .globl main
5  main:
6      pushq %rbp
7      movq %rsp, %rbp
8      subq $32, %rsp                # Locais para salvar %rsi e %rdi
9
10     movq %rdi, -24(%rbp)           # salva %rdi (argc)
11     movq %rsi, -32(%rbp)           # salva %rsi (argv[0])
12
13     movq -32(%rbp), %rbx
14     movq (%rbx), %rsi              # segundo parametro: *argv[0]
15     movq $str1, %rdi              # primeiro parametro: $str1
16     call printf
17
18     movq -24(%rbp), %rdi           # restaura %rdi
19     movq -32(%rbp), %rsi           # restaura %rsi
20
21     movq $0, %rdi
22     movq $60, %rax
23     syscall

```

Algoritmo 3.19 – Programa que imprime argv[0]

The 128-byte area beyond the location pointed to by %rsp is considered to be reserved and shall not be modified by signal or interrupt handlers. Therefore, functions may use this area for temporary data that is not needed across function calls. In particular, leaf functions may use this area for their entire stack frame, rather than adjusting the stack pointer in the prologue and epilogue. This area is known as the red zone.

O objetivo é economizar as instruções de alocação de variáveis locais em procedimentos folha¹³, e permitir que as demais funções usem este espaço para armazenar valores temporários. Porém, neste caso, se houver uma chamada a procedimento, estes valores serão perdidos.

¹³ aquelas que não contém ou que não executam chamadas a outros procedimentos

3.3.6 Segurança

A pilha já foi alvo de ataques, e foi considerada um ponto vulnerável em sistemas Unix (e consequentemente linux). Esta falha de vulnerabilidade era consequência da implementação de família de funções `getc`, `getchar`, `fgetc`, que não verificam violações do espaço alocado para as variáveis destino.

Como exemplo, veja o Algoritmo 3.20.

```

1 #include <stdio.h>
2 int main (long int argc, char **argv)
3 {
4     char s[5];
5     gets (s);
6     printf("%s", s);
7
8     return (0);
9 }
```

Algoritmo 3.20 – O perigoso `gets`

A primeira coisa interessante é que ao compilar este programa, surge uma mensagem curiosa:

```

> gcc Gets.c -o Gets
Gets.c: In function 'main':
Gets.c:5:3: warning: 'gets' is deprecated (declared at /usr/include/stdio.h:638) [-Wdeprecated-declarations]
    gets (s);
    ~
/tmp/ccGv030z.o: In function 'main':
Gets.c:(.text+0x26): warning: the 'gets' function is dangerous and should not be used.
```

O primeiro aviso diz que o procedimento `gets` foi descontinuada¹⁴. O segundo é mais grave. Diz que o procedimento `gets` é perigosa e que não deve ser usada. O porquê deste perigo é que ela não verifica os limites do vetor (que no caso tem cinco bytes). Para demonstrar o problema, veja o que ocorre com a execução onde a entrada de dados é maior do que o tamanho do vetor:

```

> ./get
12345678
12345678
>
```

Observe que foram digitados oito caracteres, e eles foram armazenados no vetor. Evidentemente eles não “cabem” no espaço alocado para eles, e podem sobrescrever outras variáveis.

Atualmente, o ambiente de execução verifica se há uma tentativa de sobreposição, impedindo um possível ataque. O primeiro artigo “popular” que citou esta vulnerabilidade foi “Smashing The Stack For Fun And Profit” de Aleph One.

Este ataque ocorre quando o usuário entra com uma quantidade de dados maior do que o tamanho da variável, sobreexcrevendo outros dados na pilha, como o endereço de retorno do registro de ativação corrente.

Se os dados digitados incluírem um trecho de código (por exemplo, o código de `/usr/bin/sh`) será injetado um trecho de código na pilha. O próximo passo envolve o comando *assembly* `ret`. Se antes desta instrução contiver o endereço do código injetado, então o código injetado será executado.

Agora, vamos analisar o que ocorreria na injeção em um processo que executa como superusuário, como por exemplo, o `ftp` em linha de comando.

Se o código injetado for o `sh`, o usuário não só executará a shell, mas a executará como superusuário!

Mais do que simples conjecturas, este método foi usado (e com sucesso) durante algum tempo. Para mais detalhes, veja [19].

Desde a publicação do artigo e das análises das falhas de segurança, foram criados mecanismos para evitar este tipo de invasão, como pode ser visto na execução do Algoritmo 3.20, com maior quantidade de dados.

¹⁴ do inglês *deprecated*, cuja tradução literal é “depreciado” (no sentido de contabilidade). Outra tradução possível aqui é “desaprovada”, que faz mais sentido aqui.

```
> ./get
123456789123456789
*** stack smashing detected ***: ./Gets terminated
123456789123456789Aborted
```

Os ataques baseados em *overflow* da pilha são uma grande ameaça à segurança, principalmente porque a maioria dos computadores está ligado à internet, e pode ser atacado remotamente.

Por isso, várias técnicas para impedir a exploração de ataques baseados em *overflow* da pilha foram estudados e implementados tanto pelos desenvolvedores de sistemas operacionais quanto pelos desenvolvedores de compiladores.

Em sistemas operacionais destacamos duas técnicas:

- aleatorização do espaço de endereçamento¹⁵. Muitos ataques baseiam-se no fato do sistema operacional utilizar endereços fixos para algumas funções de bibliotecas. Ao aleatorizar o local onde elas serão colocadas, elimina-se o problema. É uma técnica incluída em diversos sistemas operacionais incluindo linux e windows. Boas introduções ao tema podem ser encontrada em [52] e [6];
- não permitir execução na região da *stack*. Criar mecanismos para impedir execução de código na região da pilha não impede a injeção de código, mas impede os efeitos danosos. Esta técnica exige uma contrapartida em *hardware*, que foi implementada no AMD64 com o bit NX¹⁶ que desabilita a execução de código em regiões da memória virtual (veja Apêndice D.2.1.1).

Para uma análise comparativa dos métodos e sua efetividade, veja [15].

Já em compiladores, foram propostas mais técnicas que abordam tanto a prevenção quanto a detecção de *overflow*.

No caso do compilador *gcc*, a técnica adotada não previne, mas detecta a ocorrência *overflow* utilizando “canários”. Este nome é uma analogia a uma técnica usada para descobrir se há vazamentos de gás. Coloca-se um canário na área suspeita. Se ele morrer, há vazamento.

A versão utilizada pelo *gcc* é menos letal. Para detectar *overflow*, é inserido um conjunto de bytes perto de uma região crítica. Se os bytes forem alterados em algum momento, significa que houve um *overflow*. Para mais informações veja [41] e [30].

3.4 Registros de ativação no MIPS

Cada arquitetura estabelece um conjunto de padrões. Este livro adota o padrão sugerido por [51] para o AMD64. Porém outras arquiteturas sugerem outras abordagens que se ajustam melhor às suas necessidades e em função dos recursos que tem.

Por exemplo, o MIPS é uma CPU RISC que disponibiliza 32 registradores. O uso dos registradores é livre, mas convencionou-se adotar o padrão da Tabela 13.

Registrador	Uso
\$a0-\$a3	Parâmetros
\$s0-\$s7	Preservados entre procedimentos
\$t0-\$t9	Não preservados entre procedimentos
\$v0-\$v1	Valor de retorno do procedimento

Tabela 13 – Convenção de uso dos registradores no MIPS (adaptada de [23])

Esta tabela é enganosa, pois sugere que o sistema garante a preservação ou não dos registradores quando na verdade, sugere que o programador implemente esta semântica. Isto significa que o programador deve salvar os registradores \$t0-\$t9 antes de chamar um procedimento e quando implementar um procedimento, deve salvar os registradores \$s0-\$s7.

A convenção dos passos na chamada de procedimento no MIPS podem ser resumidos assim:

Passo 1 Os parâmetros são passados através dos registradores \$a0 até \$a3. Se houverem mais de quatro parâmetros, os excedentes (parâmetro 5, 6, etc.) serão passados na pilha, sendo empilhado do último para o quinto.

Passo 2 Resultados do procedimento são armazenados em \$v0 e \$v1.

¹⁵ Address space layout randomization

¹⁶ non-execute bits

Passo 3 Os registradores `$s0-$s7` terão os valores inalterados pelo procedimento chamado. Se o procedimento chamado quiser usá-los, deverá primeiro salvar seus valores, e ao final, restaurá-los.

Passo 4 Os registradores `$t0-$t9` não são preservados pelo procedimento chamado, que pode usá-los livremente. Se o procedimento chamador quiser preservá-los, então deverá salvá-los antes da chamada do procedimento e restaurá-los depois da chamada.

Passo 5 salvar os valores dos parâmetros (`$a0-$a3`), e para isto, pode-se abrir mais algum espaço (ao invés de 32, abrir $32+16=48$ bytes). Este caso é justificado somente se o procedimento chamar outros procedimentos (a convenção usa os termos "leaf procedure" para procedimentos que não usam o comando "jal" e "non leaf procedure" para aqueles que usam).

Tanto no passo 3 quanto no passo 4 não é necessário salvar todos os registradores. Basta salvar aqueles que serão usados. Porém, é comum encontrar programas (didáticos) que abrem espaço para todos os registradores. Isto não é comum em programas na prática.

O MIPS é um caso curioso. Apesar de existir uma padronização de uso dos registradores, não padroniza a organização do registro de ativação. Isto deu origem a implementações diferentes em compiladores.

Uma análise mais detalhada do MIPS foge ao escopo deste livro. Porém é interessante observar que existe um registrador (*frame pointer* - `$fp`) com a mesma funcionalidade do `%rbp` no AMD64. Apesar disso, ele nem sempre é usado para acessar as variáveis e parâmetros, que são acessadas a partir do registrador `$sp`. Aliás, isto também é uma opção no AMD64, o que libera um registrador (`%rbp`) para outros usos. Veja Exercício 3.10

Exercícios

- 3.1 As arquiteturas modernas não permitem leitura ou escrita no apontador de instruções (`%rip` no AMD64). Para tal, disponibilizam duas instruções que permitem que o valor de `%rip` seja empilhado e desempilhado. O problema é que isto atrapalha, mas não é um impedimento categórico. Mostre como armazenar o valor corrente de `%rip` em um registrador e como escrever qualquer valor em `%rip` usando estas duas instruções.
- 3.2 Explique por que não aparece o estágio (3) na Tabela 6.
- 3.3 O aluno Desa Tenta cometeu um erro ao copiar o programa do Algoritmo 3.7. Na linha 5, ele escreveu `z = x;` e surpreendeu-se que ao retornar do procedimento, o valor de `B` era `0x60 1048`. Explique o porquê disto.
- 3.4 Após a aula, os alunos Um e Dois (não são seus verdadeiros nomes) digitaram o programa do Algoritmo 3.19. Sentados lado a lado em computadores diferentes no laboratório, simularam a execução do programa no `gdb` e perceberam algo muito curioso. Os valores hexadecimais de `%rsp`, desde o rótulo `_start` eram diferentes quando executados na área de Um e de Dois. Perguntaram o porquê disto ao professor sugerindo que o problema eram os computadores serem diferentes, mas o professor respondeu enigmaticamente que a altura inicial da *stack* depende também da quantidade de variáveis de ambiente utilizadas em cada área de trabalho assim como a quantidade de caracteres usada em cada variável. O que isso significa?
- 3.5 Para cada um dos programas *assembly* apresentados faça:
 - a) gere o executável;
 - b) execute passo a passo usando o `gdb`. Desenhe a memória virtual indicando os endereços que o `gdb` indicar para as variáveis em cada registro de ativação, assim como as informações de contexto.
- 3.6 No programa do Algoritmo 3.9, o primeiro parâmetro da chamada de procedimento da linha 7 causa confusão aos programadores iniciantes na linguagem C. Explique o efeito de usar `*res`, `&res` em lugar de `res` usando a Figura 12. Complemente com `**res` e `&*res`.
- 3.7 Considere o programa *assembly* no Algoritmo 3.10. Escreva à direita de cada linha (ou conjunto de linhas) a quais os comandos (e linhas) do Programa 3.9 elas estão associadas.
- 3.8 A Seção 3.3.6 apresentou o mecanismo de exploração de falha da pilha para executar um programa externo. A isso se dá o nome de "injeção de código" (*code injection*). Desenhe a na execução do programa do Algoritmo 3.20 para uma entrada grande.
 - a) O que é sobrescrito?
 - b) Variando a quantidade de caracteres digitados, indique onde está localizado o canário.
- 3.9 Conforme descrito na Seção 3.3.3, o registrador `%rbx` é preservado pelo procedimento chamado. Porém, isto não foi aplicado nos programas usados neste livro. Implemente esta alteração em todos os programas apropriados.

- 3.10 Variáveis e parâmetros são acessadas a partir do registrador `%rbp`. O mesmo livro que sugere esta abordagem também indica que pode ser usado o registrador `%rsp` para a mesma tarefa [51]. Descreva o que muda no projeto dos registros de ativação se for usado o `%rsp`. É necessário guardar o valor anterior de `%rsp` na pilha da mesma forma que se faz com `%rbp`?
- 3.11 Traduza o Algoritmo 3.18 para *assembly*.
- 3.12 Execute o Algoritmo 3.18 algumas vezes:
- Os endereços dos parâmetros são sempre os mesmos? Explique.
 - Isto vale para os programas *assembly* também? Explique.
- 3.13 O que está localizado após `argv[argc]`?
- 3.14 No Algoritmo 3.21, a variável local `contador` foi declarada com o modificador `static`. Esta declaração preserva o valor da variável entre chamadas de procedimento [4]. Quando executado, o programa gera o seguinte resultado:

```
> gcc staticLocal.c -o staticLocal
> ./staticLocal
1 2
```

Deduzo como estas funcionalidade foi implementada.

```
1  #include <stdio.h>
2
3  int f()
4  {
5      static int contador = 0;
6      contador++;
7      return contador;
8  }
9
10 int main()
11 {
12     printf("%d ", f());
13     printf("%d \n", f());
14     return 0;
15 }
```

Algoritmo 3.21 – Modificador `static` em variáveis locais

4. Chamadas de Sistema

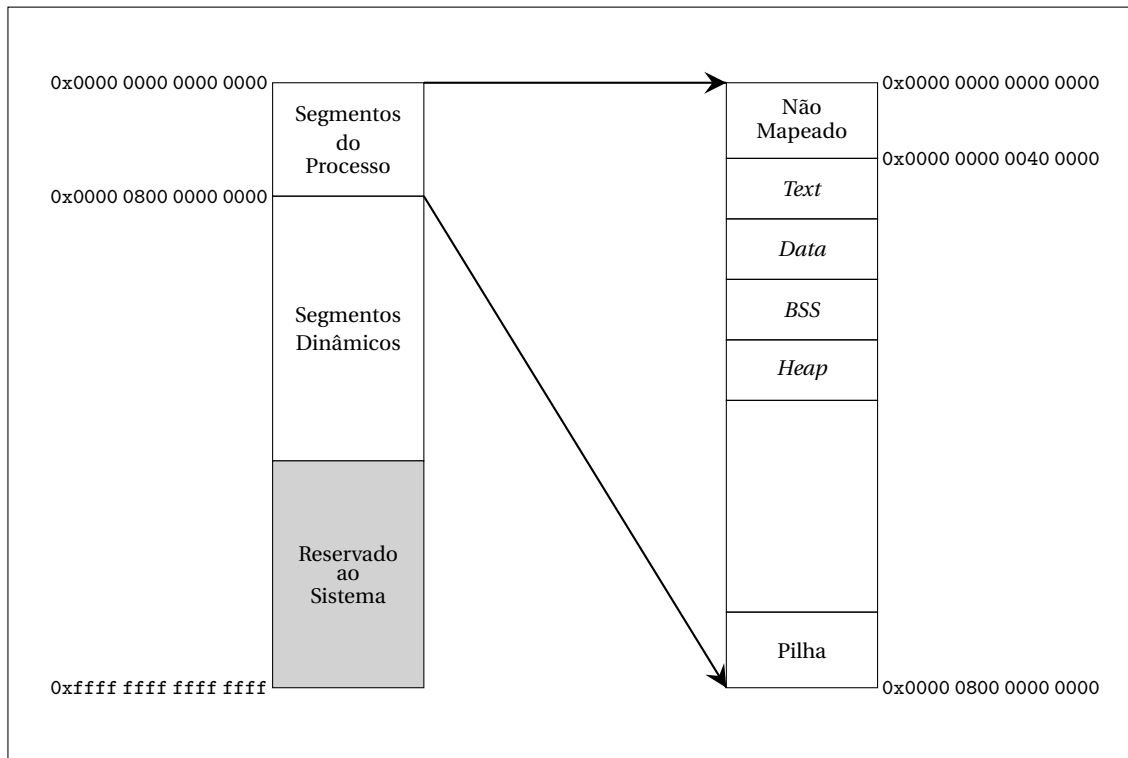


Figura 14 – Espaço Virtual de um Processo: A região reservada ao sistema

Diferente dos capítulos anteriores, a região reservada ao sistema não é uma parte dos segmentos de processo como indicado na figura 14.

Como apresentado no apêndice D o espaço virtual de endereçamento é linear, e permite estabelecer proteções a determinadas regiões. Na região dos segmentos de processo, por exemplo, é permitida a leitura e escrita nas páginas alocadas às seções *data*, *stack* e *heap*, mas não é permitida a execução. Por outro lado, é permitida a execução e leitura na seção *text*, mas não a escrita.

Em um sistema operacional decente, os processos dos usuários não devem acessar diretamente os recursos do computador como o disco ou a área de outros processos, entre outros. Desta forma, os processos tem atuação restrita, e atuam no que se chama “modo usuário”. Por outro lado, o sistema operacional tem acesso a TODOS os recursos do computador e é executado no que se chama “modo supervisor”.

Nas páginas da região reservada ao sistema, não é permitido leitura, escrita ou execução para processos atuando em modo usuário. Porém, é permitida a leitura, escrita ou execução para processos atuando em modo supervisor.

Como o próprio nome sugere, a região reservada ao sistema é o local onde o código do sistema operacional é mapeado. Todos os processos em execução tem esta região no seu espaço de endereçamento virtual, e as páginas virtuais lá contidas são mapeadas nas mesmas páginas físicas (como a página as páginas $PV0_i$ e $PV2_j$ mapeadas na PF6 na seção D.2.1.1). Com isso uma única cópia do sistema operacional fica na memória física e todos os processos em execução os mapeiam em sua região reservada ao sistema.

Quando um processo precisa de um recurso externo, ele utiliza uma instrução assembly específica (*syscall*) que:

1. muda o modo de execução de “modo usuário” para “modo supervisor”;
2. desvia o fluxo de execução para o código do sistema operacional (a área cinza da figura 14).

Quando o sistema operacional finalizar sua tarefa, utiliza uma instrução específica (*sysret*) que:

1. muda o modo de execução de “modo supervisor” para “modo usuário”;
2. desvia o fluxo de execução para a instrução seguinte ao `syscall` (a seção `.text` já estudada).

Uma vez explicado o conteúdo da região reservada ao sistema, restam dois problemas:

1. Por que acessar esta região.
2. Como acessar esta região.

Por que acessar esta região

Os computadores que adotam o modelo von Neumann são, conceitualmente, divididos em três partes: CPU, memória e dispositivos de entrada e saída (veja figura 34 na página 177).

Os capítulos anteriores deste livro só apresentaram programas que usam memória e CPU, não explicando como um processo acessa os recursos que não fazem parte de seu espaço de endereçamento como arquivos, eventos do mouse e teclados, etc., que denominamos “recursos externos”.

Permitir que qualquer processo acesse livremente os recursos externos pode causar danos graves. Tal processo malicioso poderia danificar estes recursos, por exemplo, apagando todo o disco.

Para evitar tais problemas, sistemas operacionais modernos só permitem que os processos acessem a memória alocada a eles no espaço virtual de endereçamento (com restrições no acesso a determinadas páginas) e impedem qualquer acesso aos recursos externos.

Para acessar um recurso externo, um processo deve fazer esta solicitação ao sistema operacional que é quem “suja as mãos” com os detalhes físicos para fazer o acesso e disponibilizar os dados. Como exemplo, considere que um processo deseja ler o conteúdo de um arquivo. Como ele não tem acesso aos recursos externos, ele também não tem acesso ao arquivo, mas pode pedir ao sistema operacional que atue como intermediário. O processo avisa ao sistema operacional o nome e qual região do arquivo deseja ler, e o sistema operacional executa as seguintes operações:

1. Acessa o arquivo solicitado.
2. Copia a região solicitada do arquivo para dentro do espaço de segmentos do processo;
3. Quando termina, avisa o processo que os dados pedidos estão disponíveis.

Desta forma, o processo consegue acesso a todos os recursos externos, e só trabalha com cópias deles em seu espaço virtual de armazenamento (na memória) sem sequer saber como o acesso foi feito. Toda a atividade de acesso físico é feita única e exclusivamente pelo sistema operacional.

Em linux quase todos os recursos externos são mapeados em arquivos (por exemplo, o arquivo `/dev/input/mouse0` é onde se pode “ler” as ações do usuário no mouse). Assim, o processo descrito acima é executado para acessar quase todos os dispositivos de entrada e saída.

As funções desempenhadas pelo sistema operacional não se resumem ao acesso a dispositivos externos. Elas incluem funcionalidades como criação e término de processos, criação e término de fluxos de execução (*threads*) entre vários outros.

Como acessar esta região:

Agora vamos abordar a segunda questão: como acessar esta região.

Todas as CPUs modernas incluem mecanismos para que um processo atue em um de dois “modos de operação”¹: modo usuário e modo supervisor. Um processo no modo usuário tem algumas restrições enquanto que um processo no modo supervisor tem acesso irrestrito a todos os recursos do computador.

Um determinado processo pode mudar o seu modo de operação através de instruções específicas. No AMD64 a instrução que muda de modo usuário para supervisor é a instrução `syscall` e a que muda de modo supervisor de volta ao modo usuário é a instrução `sysret`.

A instrução `syscall` faz mais do que só mudar o processo de modo usuário para supervisor. Após efetuar esta mudança, ela salva o `%rip` (para saber para onde retornar com a instrução `sysret`) e armazena em `%rip` um endereço específico da região reservada ao sistema, semelhante ao que ocorre na instrução `call`.

O mecanismo que usa a instrução `syscall` e o que usa a instrução `call` são semelhantes. Para diferenciá-los o segundo é chamado de chamada de procedimento, e o primeiro é chamado “chamada ao sistema”².

Os dois mecanismos também se assemelham no que se refere aos parâmetros: em ambos os casos prioriza-se a colocação de parâmetros em registradores. Porém são muito diferentes na forma de invocar.

Enquanto em chamadas de procedimento utiliza-se o nome do procedimento para invocá-lo, em chamadas de sistema os nomes são associados a números. Coloca-se o número do procedimento a ser executado no registrador `%rax`

¹ alguns têm mais modos de operação, mas ficaremos só em dois nesta explicação

² *system call*

antes da instrução `syscall`. Aliás, não se usa o termo “nome do procedimento” no contexto de chamadas de sistema, mas sim “serviço” o que deixa claro que uma chamadas de sistema nada mais é que a requisição que o sistema operacional execute um serviço para o processo.

Sistemas operacionais diferentes disponibilizam serviços diferentes. Os serviços disponibilizado no linux seguem a especificação POSIX³, uma família de normas definidas pelo IEEE para a manutenção de compatibilidade entre sistemas operacionais e designada formalmente como norma IEEE 1003 [7].

Vale a pena mencionar que o POSIX faz mais do que especificar a lista de serviços de um sistema operacional. Ela propõe uma API que contém variáveis de ambiente, estruturas de dados e muito mais.

Por exemplo, arquivos são mapeados como uma estrutura apontada por um descritor de arquivo⁴. Esta estrutura é codificada dentro do código do linux (em uma `struct`)⁵ e os serviços podem interagir com ela. Em nenhum momento o processo (em modo usuário) pode escrever diretamente estas estruturas, só trabalhar com cópias locais.

Explicar o que são e como funcionam as chamadas ao sistema na prática envolve conhecimentos maiores de *hardware*, e em cursos de computação são explicadas numa ou mais disciplinas específicas (normalmente denominadas “sistemas operacionais”). Para manter a abordagem deste livro mais “leve”, apresentamos algumas analogias no lugar dos detalhes exatos.

A primeira analogia é descrita na seção 4.1, e explica os modos de execução de um processo, usuário e supervisor. Em seguida, a seção 4.2 acrescenta CPU e a seção 4.3 acrescenta os periféricos (dispositivos de *hardware*) à analogia.

Com a analogia completa, a seção 4.4 explica como as coisas ocorrem na prática em uma arquitetura. Em seguida, a seção 4.6 apresenta uma outra forma de implementar as chamadas ao sistema (MS-DOS), e relata alguns problemas que podem ser gerados com esta implementação. A seção 4.5 apresenta os *drivers* nativos. Por fim, a seção 4.7 apresenta as chamadas ao sistema no linux.

4.1 A analogia do parquinho

Uma analogia que eu gosto de usar é a de que um processo em execução em memória virtual é semelhante a uma criança dentro de uma “caixa de areia” em um parquinho de uma creche.

Imagine uma creche onde as crianças são colocadas em áreas delimitadas para brincar, uma criança por área. Normalmente isto acontece em creches, onde estas áreas são “caixas de areia”.

Porém nesta analogia, ao contrário do que ocorre em creches, cada criança é confinada a uma única área, não podendo sair dali para brincar com outras crianças.

Agora, considere uma criança brincando em uma destas áreas. O que ela tem para brincar é aquilo que existe naquela caixa de areia. Ali, ela pode fazer o que quiser: jogar areia para cima, colocar areia na boca (e outras coisas que prefiro não descrever).

Tudo o que a criança fizer naquela caixa terá consequências unicamente para ela, ou seja, não afeta as crianças em outras caixas de areia.

Quando a criança terminar de brincar, a caixa de areia é destruída. Para cada criança que quiser brincar mas não estiver em uma caixa de areia, uma nova caixa será construída para ela, possivelmente usando areia de caixas destruídas anteriormente.

Porém, é importante garantir que as “lembranças” deixadas por uma criança não sejam passadas para outras crianças. Uma forma de fazer isso é esterilizando todo o material das caixas de areia destruídas, em especial a areia utilizada.

Quem faz esta tarefa é a “tia”, que aqui é conhecida como “supervisora”. Aliás, é bom destacar que é isto que se espera de uma creche bem conceituada: que cuide da saúde das crianças. Você deixaria seu filho em um parquinho onde a supervisora negligencia a saúde das crianças?

Esta supervisora também é responsável por outras tarefas, como atender às requisições das crianças por dispositivos externos.

Suponha que existem brinquedos para as crianças usarem nas caixas de areia: baldinho, regador, etc. Como as crianças não podem sair de sua caixa de areia, elas têm de pedir o brinquedo para a supervisora.

A supervisora recebe o pedido, e procura por algum brinquedo disponível. Eventualmente irá encontrar (nem que seja tirando de uma caixa de areia onde está outra criança), porém antes de passar para a criança que solicitou, deve primeiramente higienizar o brinquedo para que lembranças deixadas pela primeira criança não afetem a segunda criança.

³ *Portable Operating System Interface*

⁴ *file descriptor*

⁵ lembre que o linux é composto quase totalmente de código em escrito em linguagem C.

Nesta analogia, temos:

- crianças e supervisora são os processos;
- crianças têm uma área restrita de ação, enquanto que a supervisora tem uma área de atuação quase total;
- cada caixa de areia é a memória virtual de um processo.
- os brinquedos são recursos, como por exemplo, dados de arquivo, de teclado, de mouse, etc. Ou seja: são os objetos que um processo não pode acessar porque estão fora de sua memória virtual;
- os pedidos e devoluções de recursos externos são executados através das chamadas ao sistema (*system calls*).

Um detalhe importante é que crianças e supervisores, apesar de serem processos, atuam em “modos” diferentes. Enquanto uma criança só pode acessar a sua caixa de areia e as coisas que estiverem lá dentro, a supervisora pode acessar qualquer caixa de areia, os espaços entre as caixas de areia e até fora delas (como por exemplo, em sua sala particular). Estes dois modos são conhecidos como “modo usuário” (para as crianças) e “modo supervisor” (para a supervisora).

Assim, as chamadas ao sistema são basicamente requisições que o processos fazem ao sistema operacional por recursos externos à sua área virtual. Para acessar qualquer coisa que estiver fora de sua área virtual, o processo deve fazer uma chamada ao sistema.

4.2 Acrescentando a CPU

Na analogia da seção anterior, todos os “processos” (crianças e supervisora) podem trabalhar em paralelo, ou seja, duas crianças podem brincar em suas caixas de areia simultaneamente.

Porém, em uma arquitetura real, somente os processo que estão usando a CPU é que estão ativos. Os demais estão “em espera”.

Para explicar isso na analogia do parquinho, vamos introduzir um pouco de magia.

Uma moradora vizinha não gosta do barulho que as crianças fazem quando estão brincando. Para azar de todos, ela é uma bruxa malvada que invocou uma maldição terrível para silenciar todos.

Esta maldição ilumina todo o parquinho com uma luz cinza que transforma todos os atingidos em estátuas (inclusive a supervisora).

Outra vizinha é uma bruxa boazinha, que estranhando a falta de barulho, percebeu o que ocorreu. Como ela não é uma bruxa tão poderosa quanto a bruxa malvada e não pôde reverter o feitiço.

Mesmo assim, conseguiu conjurar uma fadinha voadora, imune à luz cinza. Quando ela sobrevoa uma criança (ou a supervisora) consegue “despertá-la” por algum tempo usando uma luz colorida. Infelizmente, a fadinha só consegue acordar exatamente uma pessoa por vez.

Por esta razão, a bruxa boazinha pediu para a fadinha não ficar muito tempo em uma única pessoa, e mudar de pessoa a pessoa para que todos pudessem brincar um pouco.

Porém, a bruxa boazinha observou que a fadinha não pensava muito para escolher a próxima pessoa para acordar, e com isso ela acabava acordando somente um grupo pequeno de pessoas (e outras não eram sequer acordadas).

Para encontrar uma alternativa mais justa, a bruxa boazinha pediu para que a fadinha sempre acordasse a supervisora após cada criança. A supervisora foi informada da situação, e foi incumbida de encontrar uma forma mais justa para encontrar a próxima criança para acordar.

Toda vez que a fadinha acordava a supervisora, ela anotava qual a última criança acordada e escolhia a próxima a ser acordada de acordo com as suas anotações. Em seguida, a fadinha ia acordar a criança indicada pela supervisora.

Outro problema é que a fadinha não tinha relógio. Com isso, por vezes ficava tempo demais sobre uma criança antes de voltar à supervisora, causando injustiças.

Para resolver este último problema, a bruxa boazinha deu um *pager* à fadinha, e instalou um relógio que ativa o *pager* em intervalos regulares. Estes intervalos são chamados de “quanta”, e correspondem ao tempo máximo que a fadinha fica sobre uma criança.

Cada vez que o relógio ativa o *pager*, a fadinha acorda a supervisora e mostra o *pager* para ela. Lá está indicado quem é que o ativou (no caso, o relógio). Com esta informação, a supervisora sabe que deve executar uma tarefa específica: escolher qual a próxima criança ser acordada. Quando fizer a escolha, a fadinha vai até a criança indicada e a ilumina até que ocorra outra ativação do *pager*. Outros dispositivos também podem ativar o *pager*, como será visto adiante.

Imagine agora que o intervalo do relógio é de um minuto. Quando se passar uma hora, até 60 crianças terão brincado. Se o intervalo do relógio for menor, digamos, um milissegundos (1ms), um observador externo terá a ilusão que todas as crianças estão brincando **em paralelo**. É isto que ocorre em um computador. O intervalo de execução de cada processo é tão pequeno, que em um segundo, dezenas de processos entram e saem de execução, dando a ilusão de paralelismo. A esta ilusão se dá o nome de “pseudoparalelismo”.

A fadinha que foi apresentada aqui corresponde à CPU. A alternância de processos (*process switch*) ocorre da forma sugerida aqui. Cada vez que o relógio bate, a CPU consulta o sistema operacional para que ele indique qual o próximo processo a ser acordado. Existem vários algoritmos para escolher qual o próximo processo a ser acordado, como por exemplo, escolher o processo que está há mais tempo sem utilizar a CPU.

4.3 Acrescentando periféricos e dispositivos de hardware

Outra entidade que será estendida é aquela que fornece os brinquedos. Imagine que do lado de fora do parquinho existem vários quiosques especializados em fornecer brinquedos. Um quiosque só guarda baldinhos, outro só regadores, e assim por diante. Quando uma criança chama a supervisora, ela pode especificar qual o brinquedo que ela quer. Por exemplo, “quero o baldinho azul turquesa, número 0xab00” (sim, as crianças sabem hexadecimal). A supervisora anota o número envia uma mensagem para o quiosque de baldinhos solicitando o baldinho correspondente. Em sua caderneta, a atendente também anota qual foi a criança que pediu por aquele baldinho.

O atendente de cada quiosque é meio “lento”, pois são dezenas de milhares de brinquedos em cada quiosque (não me perguntem como eles cabem lá). Cada quiosque tem uma fadinha própria (ou seja, a fadinha do parquinho não precisa ir até lá para dar vida ao atendente, pois ele nunca vira estátua).

Se a supervisora ficar esperando pelo baldinho, ela não poderá atender às demais requisições das crianças, causando grande consternação audível (mesmo sendo de uma só criança por vez). Por esta razão, ela volta a atender as requisições do parquinho e espera que o atendente avise quando encontrar o que foi solicitado.

Quando o atendente encontrar, ele precisa avisar à supervisora que encontrou o que foi pedido. Para este tipo de situação, a fadinha utiliza novamente o *pager*, que é acionado pelos atendentes dos quiosques. Ao perceber que o *pager* foi acionado, a fadinha vai até a supervisora e a acorda. No *pager* tem o número do quiosque que enviou a mensagem. A supervisora vai até o quiosque e pega o dispositivo. Em seguida, procura em sua caderneta quem pediu aquele dispositivo, e o coloca na caixa de areia correspondente.

Como o mecanismo de pegar brinquedos pode ser diferente dependendo do quiosque (por exemplo, para pegar um baldinho usa-se um mecanismo diferente do que para pegar areia, ou água), o número do quiosque já é suficiente para que a supervisora saiba como pegar aquele brinquedo. O atendente nunca sai do quiosque, e por isso, um mensageiro dedicado àquela tarefa é usado para transportar o baldinho. Para cada quiosque há um mensageiro especializado.

Nesta versão estendida da analogia, temos que:

- as fadas correspondem à CPU. Existem tantas fadas para dar vida ao parquinho quantas núcleos em uma CPUs.
- os quiosques são os dispositivos periféricos (disco, teclado, mouse, etc.). Quase todos têm uma CPU dedicada a executar o que foi requisitado pela supervisora. Porém, eles não tem acesso ao parquinho.
- quando a requisição ao quiosque fica disponível, o atendente ativa o *pager*, que em termos técnicos é chamado *interrupção*. Uma CPU sabe que houve uma interrupção quando um pino específico (normalmente chamado de INTR) é ativado.
- a supervisora é um processo. Ela executa código, e tem “sub-rotinas” dedicadas a tratar as requisições dos dispositivos. Cada dispositivo pode ser tratado por um trecho de código diferente, e uma forma de acessar o trecho de código correto é através do número que aparece no *pager*. Após perceber que o pino INTR foi ativado, a CPU “pergunta” quem mandou a interrupção, e o dispositivo responde com o seu número.

É importante observar que a supervisora só pode fazer as atividades descritas em uma lista de serviços. Estes serviços incluem buscar baldinho, mas não inclui comprar cachorro quente, por exemplo. A lista de serviços aos quais um sistema operacional responde varia de um S.O. para outro.

4.4 Interrupções

A analogia da seção anterior é bastante fiel ao que realmente ocorre em um computador para que os processos “conversem” com os dispositivos. A conversa ocorre através das chamadas de sistema, e esta seção aumenta o nível de detalhe do que ocorre dentro de uma arquitetura. Cada arquitetura trata estes eventos de maneiras diferentes, e aqui apresentaremos como funciona o modelo de tratamento das arquiteturas da família AMD64.

A “conversa” entre um processo e os dispositivos externos envolve dois passos. O primeiro ocorre quando o processo solicita dados ao dispositivo, e o segundo ocorre quando o dispositivo coloca os dados na área virtual do processo.

Por uma questão didática, descreveremos inicialmente o segundo passo e depois o primeiro.

A seção 4.4.1 explica como trazer os dados de um dispositivo externo (um quiosque) para dentro da área virtual de endereçamento de um processo (a caixa de areia). Esta ação é executada utilizando um mecanismo chamado de “interrupção de *hardware*”.

Por fim, a seção 4.4.2 descreve as “interrupções de *software*” e uma delas em especial, aquela que executa as chamadas ao sistema.

4.4.1 Interrupção de *Hardware*

Considere que um dispositivo de *hardware* terminou uma tarefa. Por exemplo, que o disco acabou de ler o bloco de dados que lhe foi solicitado. Neste ponto, o dispositivo (no caso, disco) envia um sinal para a CPU. Na maioria dos computadores, este sinal ativa um pino específico chamado “INTR”. A este evento dá-se o nome de “interrupção”.

Quando este pino é ativado, a CPU termina a instrução que está executando, e suspende a execução para tratar a interrupção. Porém, como todos os dispositivos estão ligados no mesmo pino, a CPU só sabe que alguém chamou, mas ainda não sabe qual foi o dispositivo que ativou a interrupção.

Para descobrir, a CPU “pergunta” quem foi que disparou a interrupção, e o dispositivo que o fez envia o seu número para a CPU (cada dispositivo tem um número diferente).

Agora, a CPU já sabe quem ativou o pino, porém não há uma forma padronizada para “buscar” os dados que estão lá. Por exemplo, a forma de recuperar informações de um pendrive é diferente de recuperar informações do mouse. Cada dispositivo tem uma forma diferente de ser acessado (alguns têm até uma linguagem própria). Para recuperar os dados, é necessário saber qual o dispositivo para saber como acessá-lo usando a forma correta (ou linguagem correta).

Por causa disto, o sistema operacional contém uma lista de procedimentos projetados para acessar os dispositivos de acordo com suas especificidades. Cada procedimento trata uma (ou mais) interrupção de hardware, e é conhecido como “*driver*” do dispositivo.

Como existem vários *drivers*, a CPU precisa ativar o *driver* correto para cada interrupção. Uma forma de implementar isto é usar o número do dispositivo como índice em um vetor, denominado “vetor de interrupções” que contém os endereços dos procedimentos de cada *driver*. Este mecanismo parece complicado à primeira vista, e para deixá-lo mais claro, vamos dividir sua descrição em duas fases:

1. Colocar os endereços dos *drivers* no vetor de interrupções. Este processo é chamado de “registro de *driver*”.
2. Usar o vetor de interrupções para disparar o *driver* correto;

4.4.1.1 Registrando o *driver*

O algoritmo 4.1 mostra como armazenar os endereços de cada *driver* no vetor. Isto é feito durante o carregamento do sistema operacional (*boot*) para a memória.

Neste exemplo, o procedimento `driver0()` corresponde ao trecho de código do sistema operacional que faz o tratamento de interrupção do dispositivo de número zero.

Na prática, nem todos os *drivers* precisam ser carregados, uma vez que nem todos os dispositivos estão presentes em algumas máquinas.

```
1  ...
2  driver0 () { ... }
3  driver1 () { ... }
4  ...
5  driverN () { ... }
6  ...
7  alocaDriverEmVetor () {
8      ...
9      vetorInterr[0] = &driver0;
10     vetorInterr[1] = &driver1;
11     ...
12     vetorInterr[N] = &driverN;
13     ...
14 }
```

Algoritmo 4.1 – Carregamento dos endereços dos *drivers* no vetor

4.4.1.2 Disparando o *driver* correto

O algoritmo 4.2 mostra esquematicamente uma rotina de encaminhamento de interrupção de *hardware*. O seu parâmetro é o número do dispositivo.

`driverCerto` é uma variável que armazena o endereço de um procedimento. Basta copiar o endereço contido no vetor de interrupção para esta variável e dispará-la para que o *driver* apropriado seja executado.

```
1  ...
2  void TrataInterrupHardw (int numeroDispositivo) {
3      void (*driverCerto)();
4      ...
5      driverCerto = vetorInterr[numeroDispositivo];
6      driverCerto ();
7      ...
8  }
```

Algoritmo 4.2 – Direcionamento para o driver correto

O algoritmo 4.2 é um modelo conceitual. Na prática, existem algumas diferenças:

- o vetor de interrupções não é uma variável do sistema operacional, e sim um conjunto de endereços fixos na memória. Nos processadores da família x86, por exemplo, o início deste vetor é configurável [34] porém como referência, considere que começa no endereço 0x0 (zero absoluto da memória);
- quando um dispositivo dispara uma interrupção, ele envia o seu número para a CPU. Ao receber o número do dispositivo, a CPU usa este número como índice do vetor de interrupção, e coloca o endereço lá contido no `%rip`.

Em linux, o elenco de interrupções é armazenado no arquivo “/proc/interrupts”, e pode ser visualizado da seguinte forma:

```
> cat /proc/interrupts
```

A explicação do resultado obtido foge ao escopo deste livro, mas pode ser facilmente encontrada através de buscadores da internet.

É importante destacar que nem todas as entradas do vetor de interrupção são preenchidas. Considere que temos 10 dispositivos de *hardware*. Eles não precisam ser mapeados entre os índices [0..9], podendo ser mapeados, por exemplo, entre os índices [0..19]. Isto implica dizer que podem existir “buracos” no vetor de interrupção.

4.4.1.3 Exemplo

As interrupções de *hardware*, como o próprio nome já diz, são para atender às requisições dos dispositivos de *hardware*. Alguns deles são de utilidade óbvia como disco, mouse, teclado, vídeo, entre outros.

Porém existem alguns cuja utilidade não é aparente. Um destes dispositivos é o *timer*. Este dispositivo gera um impulso em intervalos regulares (como o relógio da seção 4.2).

Este dispositivo é um oscilador de cristal que gera interrupções em intervalos regulares, e é tratado como interrupção de *hardware*⁶.

A interrupção que este dispositivo gera não é usada para transferir dados, mas sim para “acordar” o sistema operacional, que então pode realizar tarefas administrativas, como por exemplo, trocar o processo em execução por outro processo.

É por isso que quando a “fadinha” da seção 4.2 ouve o barulho do relógio, ela ilumina o supervisor para que ele indique uma nova criança para a fada iluminar.

Normalmente, quando o timer dispara o sistema operacional, a primeira tarefa do S.O. é desabilitar as interrupções, para que ele não possa ser incomodado durante sua execução.

4.4.2 Interrupção de *Software*

A seção anterior mostrou que o vetor de interrupções é alimentado com os números dos dispositivos. Porém, também é possível colocar endereços de *drivers* que não estão associados com dispositivos, mas sim com eventos de *software*.

⁶ Procure a posição deste evento no vetor de interrupções no arquivo /proc/interrupts.

Os *drivers* contidos nestes índices evidentemente não serão ativados por eventos de *hardware*, mas sim através do que se chama de “interrupções de *software*” ou por “armadilhas”⁷. O primeiro termo normalmente é usado para indicar eventos previstos pelo programa, como as chamadas ao sistema. O segundo termo é usado para eventos causados por condições excepcionais como divisão por zero, página inválida, etc.

Nos AMD64, as interrupções de *software* são ativadas pelo comando assembly `syscall` que desvia o fluxo para uma entrada específica do vetor de interrupção.

Assim como nas interrupções de *hardware*, quem alimenta o vetor interrupção com os endereços das rotinas de tratamento das interrupções e *software* é o sistema operacional durante o *boot*.

No linux, estas rotinas são parte integrante do sistema operacional, ou seja, `syscall` é uma forma de disparar a execução de um trecho de código do sistema operacional.

Como um processo regular não pode acessar a área do sistema operacional (e muito menos o vetor de interrupção), a instrução `syscall` primeiramente muda o modo de execução de “modo usuário” para “modo supervisor” e em seguida, desvia o fluxo para o endereço indicado no vetor de interrupção.

Os processadores mais modernos usam uma variação deste mecanismo. A maior parte dos sistemas operacionais usam um único endereço no vetor de interrupção (no linux do x86 era o endereço 0x80), e por isso é comum alocar um registrador só para contê-lo. Este registrador sofre uma atribuição no *boot* do sistema operacional e depois disto, só pode ser lido.

Um programa no “modo supervisor” tem acesso a todos os recursos do hardware, e pode acessar qualquer posição de memória ou periféricos.

Quando a requisição for atendida (ou enviada para o dispositivo apropriado), o sistema operacional executa a instrução `IRET` para retornar para o programa que solicitou a chamada ao sistema, em “modo usuário”.

4.5 Os *drivers* nativos

Tudo o que foi explicado até aqui parte do princípio que o sistema operacional já foi carregado, em especial os *drivers* dos dispositivos.

Quando se liga um computador, uma das primeiras tarefas executadas é buscar pelo sistema operacional nos dispositivos do *hardware* como por exemplo, no pendrive, disquete, cd, disco rígido.

Porém, se os *drivers* são parte integrante do sistema operacional (ainda não carregado), como é possível acessar os dispositivos de *hardware*?

A resposta está nos *drivers* “pré-instalados”. Estes *drivers* estão armazenados em memória ROM⁸, ou seja, estes *drivers* são parte da máquina. Sem eles, não seria possível acessar os dispositivos de *hardware*.

Abaixo, descrevemos em linhas gerais o que ocorre quando um computador é ligado. Uma descrição mais detalhada foge ao escopo deste livro e pode ser encontrada em livros como [34] ou em vários sítios na internet.

Assim que o computador é iniciado, um trecho de código assembly contido nesta memória ROM é executado. Este trecho de código faz uma série de verificações e analisa quais os dispositivos disponíveis no computador. Em seguida, o usuário pode indicar qual o dispositivo que ele quer que seja usado para iniciar o computador.

Ao saber qual o dispositivo que deve ser usado, o programa examina a MBR⁹ do dispositivo. No disco rígido de um PC, a MBR corresponde à trilha zero.

O MBR não tem grande capacidade, e contém dois tipos de informação:

Master Partition Table informações sobre as partições contidas no dispositivo.

Master Boot Code um pequeno trecho de código que será usado para carregar o sistema operacional.

Nosso interesse é o *Master Boot Code*. Este trecho é carregado para a memória física e após carregado, o controle (%rip) é passado para ele.

Ele usa os *drivers* nativos (BIOS) para acessar o dispositivo e carregar um programa (por exemplo, o sistema operacional) para a memória. Quando ele termina, o controle é passado ao programa recém carregado.

É só nesta hora que os *drivers* do sistema operacional são carregados para a memória, e os endereços do vetor de interrupção são atualizados.

Os *drivers* da BIOS são genéricos, e funcionam para vários dispositivos. Porém, alguns dispositivos requerem *drivers* específicos que são capazes de otimizar o seu uso. É o caso de *drivers* de vídeo, que atendem desde requisições simples (como iluminar um ponto na tela) até preencher uma região com uma cor específica. Para gerar melhores

⁷ *trap*

⁸ Read Only Memory

⁹ Master boot record

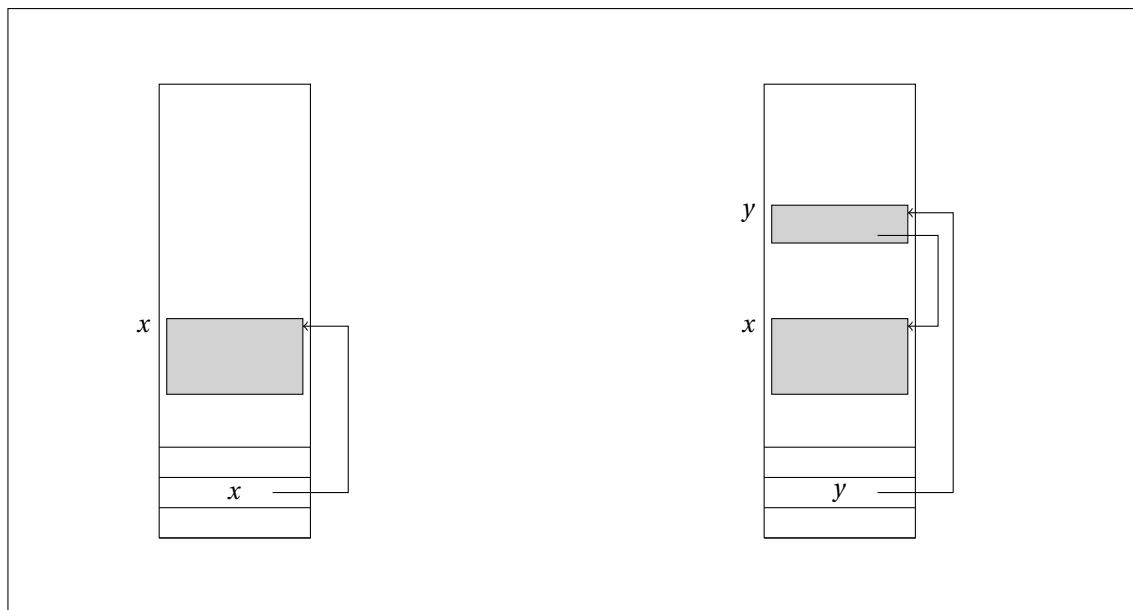


Figura 15 – Código intruso no vetor de interrupção.

resultados gráficos, e altamente recomendável que se use o *driver* apropriado àquela placa de vídeo. Normalmente este driver é fornecido pelo fabricante do hardware, e pode ser facilmente instalado.

Existem livros que explicam como escrever *drivers*, dentre os quais destacamos [36] para Windows 7 e [20] para linux.

4.6 Questões de segurança

Este livro aborda principalmente o sistema operacional linux, que é um sistema desenvolvido com recursos de segurança bem elaborados. O primeiro deles é que um processo regular não tem acesso a todos os recursos do sistema, e o único processo que tem acesso a estes recursos é o sistema operacional.

Quanto a este aspecto, vale mencionar que durante a execução de um programa,

1. somente no intervalo entre os comandos SYSCALL e SYSRET é que o sistema fica em modo supervisor;
2. durante este intervalo, o código executado é código do sistema operacional;
3. um processo não tem direito de escrita na região onde se encontra o vetor de interrupções. Com isso, ele não pode registrar um *driver* próprio.

Porém, nem todos os sistemas operacionais seguiram o mesmo modelo, causando problemas de segurança que valem a pena ser mencionados. Um dos exemplos mais conhecidos é o do sistema operacional DOS da Microsoft.

Neste sistema operacional, os processos tinham direito de leitura e escrita a toda a memória, inclusive à região do vetor de interrupção. Quando do *boot*, o sistema colocava os endereços dos *drivers* para as interrupções de *hardware* e *software* no vetor de interrupção. Qualquer programa podia alterar o vetor de interrupção, mas muitos usavam uma abordagem sutil para não parecer que havia um intruso.

Esta abordagem sutil consistia em trocar o endereço no vetor de interrupção pelo endereço da rotina-intruso. A sutileza está na última instrução da rotina-intruso: um `jmp` para o endereço contido anteriormente no vetor de interrupção, como indicado na figura 15.

A figura mostra o espaço de endereçamento virtual de um processo. As entradas da parte de baixo representam o vetor de interrupção. O lado esquerdo da figura mostra como estava uma determinada entrada do vetor de interrupção antes da alteração: a entrada apontava para o *driver* contido no endereço *x*.

O lado direito da figura mostra a situação após o intruso alterar a entrada com um novo endereço, *y*, que ao final desvia o fluxo de volta para *x*.

O código contido no endereço *y* é o intruso que será executado todas as vezes que aquela interrupção for ativada.

Observe que o serviço será realizado, e por isso não há porque suspeitar da presença de um intruso (que é uma das várias formas de instanciar um vírus no MS-DOS, onde todos os processos executam em modo supervisor).

Agora, vamos imaginar que este vírus faz algo maldoso. Por exemplo, ele apaga alguns caracteres que estão apresentados na tela, ou algo mais divertido: ele faz com que algumas das letras “caiam” como em um escorregador.

Se o usuário desligar o computador e religá-lo, o vetor de interrupções será carregado com os valores originais e o problema só voltará a existir se o usuário executar um aplicativo que injeta o driver maldoso. O problema é que alguns vírus alteravam os arquivos de configuração (executados logo após o carregamento do sistema operacional) para que estes os carregassem sempre.

Vamos supor que além de brincar de escorregador, o código maldoso também “se copia” para dentro de outros arquivos executáveis (é parte do código iniciado em *y*). Desta forma, após algum tempo, vários outros programas executáveis do sistema estarão corrompidos. Quando qualquer um destes programas for executado, o vírus irá se instalar novamente substituindo aquele *driver* e continuará infectando outros programas.

Alguns intrusos ficavam dormentes por meses, e só eram ativados em datas específicas. Destes, eu destaco dois:

1. o sexta-feira 13 apagava todos os arquivos e programas. Aliás, o nome original era “vírus Jerusalém”, pois foi detectado em Jerusalém (Israel) em 1988. Porém existem versões que dizem que ele foi criado para comemorar o 40º aniversário da criação do estado judeu¹⁰.
2. o Michelangelo (ativado no dia 6 de Março de cada ano, data de nascimento do artista), formatava o disco. Ele residia no registro de *registro* de boot do disco e infectava qualquer disquete inserido no computador.

Esta era uma realidade muito comum com o sistema operacional MS-DOS. Já encontrei computadores onde o vetor de interrupção apontava para uma cadeia de vírus.

Este exemplo mostra os problemas de segurança que podem ocorrer com o uso descuidado das interrupções.

É importante mencionar que o projeto do MS-DOS partiu do princípio que cada computador seria usado somente por uma pessoa¹¹, com programas adquiridos sempre de empresas idôneas. Em poucos anos, a realidade mudou:

1. várias pessoas passaram a usar um mesmo computador. Algumas, com veia humorística bastante desenvolvida, criaram programas para “pregar peças” em quem usaria o programa depois deles;
2. a cópia indiscriminada de programas de um computador para outro. Se um programa infectado fosse usado em outro computador, então muito provavelmente o infectaria também;
3. o uso de redes de computadores facilitou as cópias descritas acima;
4. ao contrário da premissa inicial, nem todas as pessoas que compraram computadores pessoais eram pessoas “bem intencionadas”.

Uma vantagem da abordagem do DOS era que ele permitia que cada usuário criasse sua própria rotina de tratamento para armadilhas, como divisão por zero (algo não muito útil, na verdade). Ou ainda escrever o seu próprio *driver* de tratamento de dispositivo de *hardware* (algo bastante útil). Com o surgimento de muitos dispositivos, era “fácil” criar um *driver* para cada dispositivo.

Permitir o acesso irrestrito aos dispositivos é muito perigoso, porém pode dar vez a “obras” interessantes. Exemplos incluem fazer com que controladores de disquetes reproduzam músicas como a marcha imperial de Star Wars, ou o tema do Super Mário usando *Stepper Motor*, entre várias outras.

4.7 As chamadas ao sistema no Linux

O comando que faz as chamadas ao sistema no linux já foi visto: `syscall`. Os parâmetros são passados em registradores. Como exemplo, considere a chamada de sistema que requisita o serviço `exit`:

```
> movq $60, %rax
> movq $13, %rdi
> syscall
```

A chamada ao sistema é executada pelo comando `syscall`. Os dois comandos anteriores são os parâmetros desta chamada. O valor em `%rax` indica qual serviço é pedido (terminar a execução do programa) enquanto que o valor em `%rdi` indica qual o valor a ser colocado na variável de ambiente `$?`.

¹⁰ Eu mesmo vi muita gente comemorando a plenos pulmões!

¹¹ Afinal a sigla PC vem de *Personal Computer*

4.7.1 Conjunto de serviços

Cada sistema operacional apresenta um conjunto de chamadas ao sistema, e normalmente elas não são compatíveis com outros sistemas operacionais.

O mundo Unix adota um conjunto de chamadas ao sistema denominado POSIX¹², que corresponde a um conjunto de normas definidas por um grupo de trabalho da IEEE, chamada formalmente de IEEE 1003, mas também é conhecida como norma internacional ISO/IEC 9945.

Esta normatização se fez necessária em função dos vários sistemas derivados de Unix que surgiram ao longo da década de 70 e 80. Como não havia um padrão, cada sistema era livre para adotar o conjunto que desejasse trazendo problemas de compatibilidade.

O POSIX estipula um conjunto de serviços que um processo pode solicitar ao sistema operacional.

O linux adota o padrão POSIX. Para solicitar a execução de um serviço, o número do serviço deve ser indicado no registrador `%rax`. Abaixo, descrevemos alguns destes serviços:

- 0** ler dados de arquivo (read);
- 1** escrever dados em arquivo (write);
- 2** abrir arquivo (open);
- 3** fechar Arquivo (close);
- 12** ajustar altura da heap (altera a brk);
- 60** finalizar programa (exit);

A tabela 14 descreve estes serviços com mais detalhes, como seus parâmetros e funcionalidade.

<code>%rax</code>	Nome	<code>%rdi</code>	<code>%rsi</code>	<code>%rdx</code>	Comentário
0	read	descritor de arquivo	início do buffer	tam. buffer	lê para o buffer indicado
1	write	descritor de arquivo	início do buffer	tam. buffer	escreve o buffer para o arquivo indicado pelo descritor.
2	open	ponteiro para nome do arquivo	flags	modo	cria um descritor de arquivos (em <code>%rax</code>) se o arquivo existe
3	close	descritor de arquivo			libera o descritor de arquivo indicado.
12	brk	Novo valor da brk (Se zero, retorna valor atual de brk em <code>%rax</code>)			altera a altura da brk, o que aumenta ou diminui a área da heap.
60	exit	retorno			Finaliza o programa

Tabela 14 – Exemplos de chamadas de sistema (syscalls).

O POSIX estipula por volta de duzentos serviços, e a título de curiosidade, descreveremos abaixo dois serviços especiais que permitem a criação de um novo processo:

fork Cria um novo processo. O novo processo herda várias características do processo criador, como descritores de arquivo, etc. Após a execução desta chamada de sistema, dois processos idênticos estarão em execução, e exatamente no mesmo ponto do programa pai. A diferença entre eles é que no processo pai, o retorno da chamada fork (em `%rax`) é o número do processo filho, enquanto que para o retorno do processo filho, o retorno é zero.

exec substitui o processo corrente pelo processo indicado nos argumentos. Para criar um novo processo, é necessário primeiro fazer um fork e um exec no processo filho.

dup duplica os descritores de arquivo.

Várias chamadas de sistema têm procedimentos associados na linguagem C denominadas *wrapper functions*. Exemplos incluem write, read, open, close, fork, exec, entre outras. O funcionamento destes procedimentos é simples:

¹² Portable Operating System Interface

coloca-se os parâmetros do procedimento em registradores e executa a chamada de sistema associada. Todas estes procedimentos estão documentadas nas *man pages*.

Um exemplo clássico do uso das *wrapper functions* de `fork` e `exec` descreve implementação de uma shell minimalista conforme apresentado no algoritmo 4.3, onde:

```

1 while (TRUE)
2 {
3     leComando (comando, parametros);
4     if (fork() != 0)
5         wait(&status);
6     else
7         execve(comando, parametros, 0);
8 }
```

Algoritmo 4.3 – Shell minimalista de [43]

linha 3 lê o comando digitado pelo usuário (por exemplo, `ls -l`) e o divide em nome do comando `ls` e parâmetros `(-l)`;

linha 4 cria um novo processo exatamente igual a este. Para o pai, retorna o PID do filho. Para o filho, retorna zero;

linha 5 processo pai fica bloqueado até o processo filho finalizar a execução;

linha 7 processo filho copia o código executável do comando da linha 3 para o segmento de processo. Os parâmetros são colocados na pilha. O último parâmetro do `execve` indica que não deve criar o vetor `arg`. Após criar o ambiente de execução, desvia para o rótulo `_start`.

Existe um comando para visualizar quais as chamadas de sistema que um programa executa, o comando “`strace`”. Abaixo o resultado (resumido) do que se obtém ao analisar o comando `ls` aplicado em um diretório vazio.

```

> strace ls -l
execve("/bin/ls", ["ls", "-f"], [/* 86 vars */]) = 0
brk(0)                                = 0x850000
access("/etc/ld.so.nohwcap", F_OK)     = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f6c23710000
access("/etc/ld.so.preload", R_OK)     = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, st_mode=S_IFREG|0644, st_size=149365, ...) = 0
mmap(NULL, 149365, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f6c236e8000
close(3)
```

A primeira chamada ao sistema que é executada é `execve`. Em seguida, consulta a altura da `brk` (`brk(0)`), e assim por diante. A saída completa tem mais de 100 linhas.

4.7.2 Sistema de Arquivos

E linux, o acesso a arquivos é feito através da API fornecida pelo POSIX (`open`, `close`, `read`, `write`, etc.). Esta API não entra em detalhes de implementação, não explicando *como* implementar, mas sim o *que* deve ocorrer.

Esta seção descreve (muito) sucintamente o efeito destas operações e como elas interagem com o sistema operacional.

A primeira coisa a ter em mente é que um sistema operacional é um programa implementado em uma linguagem de programação. O linux é um sistema operacional escrito em suas maior parte na linguagem C com alguns trechos escritos em assembly.

Assim, o sistema operacional armazena informações em variáveis como qualquer programa, e utiliza estruturas (`struct`) para agrupar informações.

Um exemplo de estrutura complexa é a tabela de processos. Esta tabela é um vetor onde cada entrada contém todas as informações sobre exatamente um processo. São muitas informações, mas no contexto deste livro citaremos somente tabela de páginas (usada quando o processo não estiver em execução) e os descritores de arquivo.

Posto desta forma fica mais fácil entender o que ocorre na chamada de sistema `fork`. A título deste exemplo, considere que a identificação do processo (PID¹³) é o índice do processo na tabela. Assim, o processo P_i ocupa a entrada i da tabela de processos. Desta forma, as ações da chamada de sistema `fork` são (simplificadamente):

1. Encontrar uma entrada livre na tabela de processos, j ;
2. Copiar todas as informações da entrada i para a entrada j , criando assim o processo P_j .

Neste momento, o retorno do procedimento `fork()` no processo P_i será j enquanto que o retorno da procedimento `fork()` no processo P_j será zero.

Os descritores de arquivo são armazenados dentro da tabela de processos na forma de um vetor. Cada entrada deste vetor contém informações de um arquivo aberto.

Com estas informações já é possível explicar as principais operações POSIX para lidar com arquivos listadas na tabela 14.

open encontra uma entrada livre no vetor de descritores de arquivo e associa ao nome do arquivo apontado por `%rdi` com as *flags*¹⁴ indicadas em `%rsi` e com o modo¹⁵ em `%rdx`. Esta chamada retorna em `%rax` o índice da entrada encontrada (o descritor de arquivo). Esta chamada pode ser vista como um tubo que liga o índice da tabela no arquivo indicado.

close desassocia o descritor de arquivo em `%rdi` do arquivo (tira o tubo).

read executa a leitura de dados no arquivo associado ao descritor de arquivos. Pode ser visto como ler dados na outra ponta do tubo.

write executa a escrita de dados no arquivo associado ao descritor de arquivos. Pode ser visto como escrever dados na outra ponta do tubo.

Quando um processo executa a primeira operação `open` será retornado o descritor 3 pois os descritores 0, 1 e 2 são amarrados à entrada padrão (`stdin`), à saída padrão (`stdout`) e ao arquivo de erro padrão (`stderr`).

As *wrapper functions* associadas a estas chamadas permitem seu uso na linguagem C. Veja as páginas de manual associadas com os comandos abaixo.

```
> man -s2 open
...
> man -s2 close
...
> man -s2 read
...
> man -s2 write
...
```

É importante diferenciar os procedimentos acima dos procedimentos de entrada e saída formatada (`printf()`, `scanf()`, `fopen()`, `fclose()`, etc.).

Considere só a comparação do procedimento `printf()` com a procedimento `write()`. O procedimento `printf()` não imprime diretamente o resultado no arquivo indicado, mas sim em um *buffer*. Quando o *buffer* atingir uma quantidade de bytes determinada, utiliza a procedimento `write()` para imprimir no arquivo indicado.

Isto quer dizer que os procedimentos de entrada e saída formatada são uma API construída sobre a API Posix.

4.7.3 Dispositivos Externos

Em linux, todos os dispositivos externos são mapeados para arquivos. Assim, o teclado é um arquivo, como também disco rígido, pendrive, *mouse* entre outros. Estes arquivos que correspondem a dispositivos são listados no diretório `/dev`.

Em meu computador, o diretório `/dev` contém 211 entradas. Algumas são fáceis de imaginar no que mapeiam (por exemplo `/dev/input/mouse0`), e outras nem tanto (como `/dev/input/event0`).

Em princípio todos os dispositivos poderiam ser acessados como se acessa arquivos, utilizando a API fornecida pelo POSIX. Desta forma, é possível utilizar programas que operam sobre arquivos, como por exemplo, o comando `cat`, para mostrar o resultado de uma ação sobre um dispositivo externo.

Como exemplo, considere imprimir na tela os sinais gerados pela intervenção do usuário com o mouse. Este efeito é conseguido pelo comando abaixo (o resultado só aparece quando o usuário interage com o mouse).

¹³ Process ID

¹⁴ abrir só para leitura, só leitura e escrita, só escrita, etc.

¹⁵ valor numérico dos bits *rxw*, por exemplo 755 para indicar 111101101

```
> ls -l /dev/input/mouse0
crw-r----- 1 root root 13, 32 Aug 12 07:38 /dev/input/mouse0
>
sudo cat /dev/input/mouse0 | od -t x1 -w4
00000000 09 00 00 08
00000004 00 00 38 fe
00000010 fe 18 ff 00
00000014 28 00 fe 28
00000020 01 fe 28 01
00000024 ff 28 02 ff
...
```

O comando `od` (*octal dump*) apresenta o resultado de forma legível¹⁶.

Observe o uso do `sudo`. Ele é necessário pois o “arquivo” `/dev/input/mouse0` só pode ser acessado pelo superusuário. Aliás, todos os “arquivos” de `/dev/` são acessados ou por processos executando em modo supervisor ou por grupos relacionados com os dispositivos associados.

Nem todos os arquivos podem ser visualizados desta forma, porém `/dev/input/mouse0` e `/dev/input/event2` (teclado PS2) podem.

Exercícios

- 4.1 Coloque um processo em execução e analise o arquivo `/proc/PID/maps` para ver o mapa de memória deste processo.
- 4.2 Use `objdump` e `nm` para ver o mapa de memória de um arquivo executável.
- 4.3 No algoritmo 4.3, o processo pai fica bloqueado até o processo filho terminar. Nas *shell* de linux, inserir o símbolo `&` indica que o processo pai não deve bloquear. Como acrescentar esta funcionalidade?
- 4.4 Suponha que um usuário linux abriu um terminal onde está sendo executado um *shell script*. Este usuário digita um comando que abre um aplicativo gráfico em outra janela (por exemplo, `firefox`). Se ele “matar” o terminal, o aplicativo gráfico também será finalizado. Explique, em linhas gerais, como isto é implementado baseando-se no algoritmo 4.3.
- 4.5 Acrescentando rede ao parquinho. O parquinho também tem uma caixa de correio. Para que as cartas possam ser direcionadas às crianças certas, cada uma delas pode pendurar uma meia (`socket`), como ocorre no natal, e dizer à supervisora que as mensagens que chegarem àquela meia devem ser colocadas na caixa de areia dela. Descreva como acrescentar este novo dispositivo ao parquinho. Depois disso, acrescente um mecanismo que permite que as crianças também enviem mensagem pelo carteiro.
- 4.6 Estenda a analogia para múltiplas CPUs.
- 4.7 *threads* Estenda a analogia do parquinho para que mais de uma criança possa brincar na mesma caixa de areia (considere que todas são iluminadas por fadinhas diferentes). Quais os problemas que podem ocorrer? Como preveni-los?
- 4.8 A seção 4.7.2 explicou o que são descritores de arquivo e que alguns deles são associados à entrada padrão, saída padrão e erro padrão. Explique (em linhas gerais) como alterar o algoritmo 4.3 de forma a fazer a saída padrão do processo pai seja associado à entrada padrão do processo filho.
- 4.9 É muito comum alunos utilizarem a procedimento `printf()` para depurar programas que abortam a execução por motivos desconhecidos. A ideia é imprimir os locais por onde o programa passou antes de abortar. Infelizmente, o procedimento `printf()` pode não imprimir corretamente os últimos pontos de impressão. Explique por que. Explique como usar o procedimento `fflush()` e a função `setvbuf` para resolver este problema.

¹⁶ Tente `sudo cat /dev/input/mouse0` e veja o que se entende por ilegível.

5. A Seção BSS

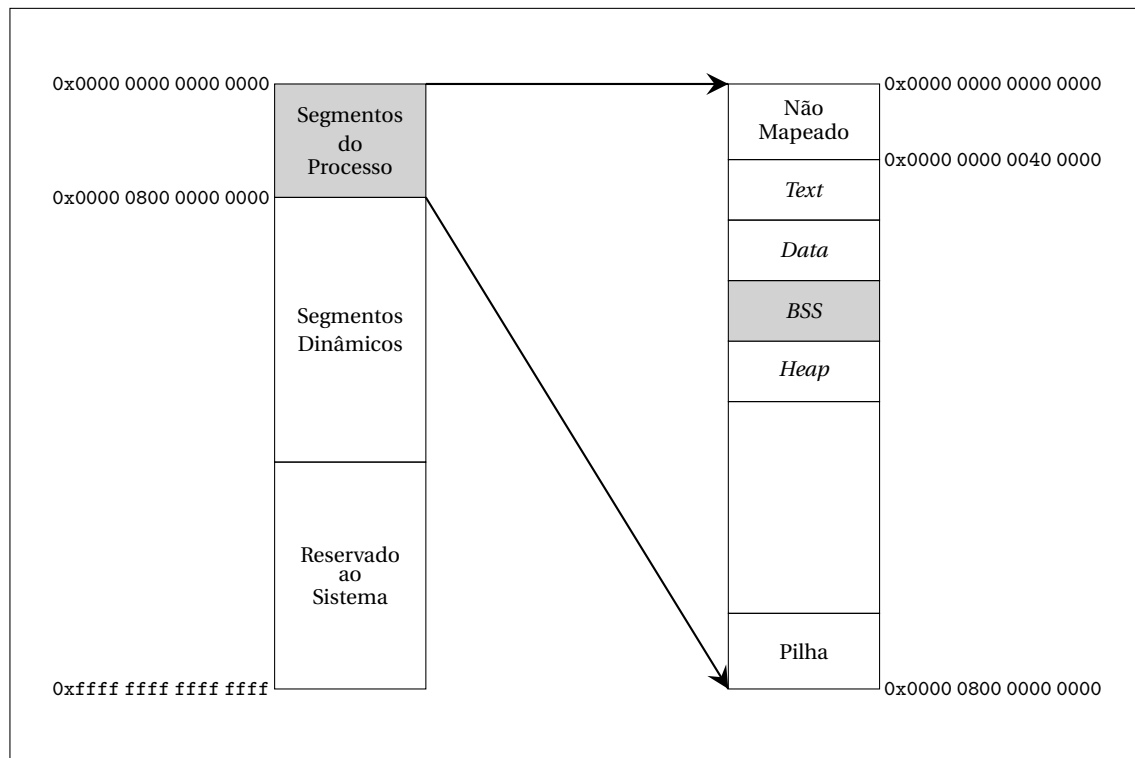


Figura 16 – Espaço Virtual de um Processo: Seção BSS

A seção da BSS (*Block Started by Symbol*) é o local para armazenar variáveis globais que utilizam grandes quantidades de dados, como por exemplo, matrizes e vetores.

Suponha um programa que lê uma matriz de 1.024 linhas e 1.024 colunas de inteiros. Em tempo de execução, este programa utilizará $1.024 \times 1.024 \times 8 = 8.3673.088$, 8Mbytes de memória só para conter esta matriz.

Se os dados forem alocados como uma variável global na seção `.data`, isto significa também que o arquivo executável teria, no mínimo, 8Mbytes, ocupando inutilmente muito espaço em disco (afinal, estes 8Mbytes só são úteis em tempo de execução).

Espaço em disco é normalmente um recurso escasso, e formas de diminuir o espaço usado (neste caso diminuir o tamanho do arquivo executável) são sempre bem-vindas. O formato ELF (assim como quase todos os formatos de execução existentes) foi projetado para esta economia através da seção `bss`. A ideia básica é que o arquivo executável contenha, na seção `.bss`, uma diretiva dizendo que, quando o programa for colocado em execução, o carregador deve alocar 8Mbytes de dados para aquela matriz. Esta diretiva ocupa poucos bytes e com isso reduz drasticamente o espaço em disco ocupado pelos arquivos executáveis. Já em tempo de execução não há quase nenhuma mudança.

Para exemplificar o funcionamento desta diretiva e a forma de utilizar a `bss`, o algoritmo 5.1 apresenta um programa C que lê um arquivo colocado no primeiro argumento e o copia para o arquivo indicado no segundo argumento. A tradução deste algoritmo é mais longa, e foi dividida em duas partes: 5.2 e 5.3.

O efeito da compilação e da execução deste programa está apresentado abaixo.

```
> gcc copiaC.c -o copia
> ./copiaC SB.ps temp
Copiando SB.ps em temp
```

```

1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <fcntl.h>
6
7  #define TAM_BUFFER 256
8  static char buffer[TAM_BUFFER];
9
10 int main ( int argc, char** argv )
11 {
12     int fd_read, fd_write;
13     int bytes_lidos, bytes_escritos;
14     printf("Copiando %s em %s \n ", argv[1], argv[2]);
15
16     fd_read = open (argv[1], O_RDONLY);
17     if (fd_read < 0 )
18     {
19         printf ("Erro ao abrir arquivo %s \n ", argv[1]);
20         return -1;
21     }
22     fd_write = open (argv[2], O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR | S_IRGRP);
23     if (fd_write < 0 )
24     {
25         printf ("Erro ao abrir arquivo %s \n ", argv[2]);
26         close(fd_read);
27         return -1;
28     }
29     bytes_lidos=read(fd_read, buffer, TAM_BUFFER);
30     while (bytes_lidos > 0 )
31     {
32         bytes_escritos = write (fd_write, buffer, bytes_lidos);
33         bytes_lidos = read(fd_read, buffer, TAM_BUFFER);
34     }
35     close(fd_read);
36     close(fd_write);
37
38     return 0;
39 }
40

```

Algoritmo 5.1 – Programa copiaC.c

```

> diff SB.ps temp
>

```

Aspectos devem ser observados:

1. Este programa utiliza a entrada e saída não formatadas (`read` e `write`) para a cópia. Estas operações são mapeadas diretamente para chamadas de sistema com o mesmo nome e seu funcionamento é descrito na seção 4.7.2.
 - A assinatura da função `open()`¹ tem três parâmetros: nome do arquivo, *flags* (no caso, abrir só para leitura ou escrita, e criá-lo se não existir) e modo (os bits "`rxw`" para usuário, grupo e outros). Os símbolos usados estão em `/usr/include/fcntl.h` e sua semântica pode ser encontrada com

```
> man 2 open .
```
2. Como alocar o vetor `buffer`. Se este vetor fosse declarado na seção `.data`, o arquivo executável deveria alocar `TAM_BUFFER` bytes para armazenar o vetor (lembre-se que as variáveis globais ocupam espaço físico dentro do arquivo executável). Para economizar espaço, é possível declarar este vetor dentro da área `bss` e dizendo quantos bytes devem ser alocados para ele quando ele for colocado em execução.
3. As constantes `O_CREAT` (crie arquivo se não existir) e `O_WRONLY` (somente para escrita) e `O_RDONLY` (somente para leitura) tem valores `0100`, `0001` e `0000` respectivamente.

¹ `int open(const char *pathname, int flags, mode_t mode);`

4. As constantes `S_IRUSR` (*read permission, owner*) e `S_IWUSR` (*write permission, owner*) indicam as permissões do arquivo e tem valores 0400 e 0200 respectivamente.

O programa assembly equivalente (Algoritmos 5.2 e 5.3) apresenta uma série de novidades:

1. Constantes: A diretiva `equ` indica uma macro, onde o primeiro conjunto de caracteres deve ser substituído pelo argumento que vem depois da vírgula. Como exemplo, todos os lugares que existe o nome `EXIT_SERVICE` serão substituídos por 60 para efeito de montagem. A ideia é que o programa se torne mais “legível”.
2. A seção `.bss` contém uma macro (`TAM_BUFFER`) equivalente ao programa em C e em seguida o comando `.lcomm BUFFER, TAM_BUFFER` que indica que, em tempo de execução, deverão ser alocados 256 bytes ao rótulo `BUFFER`. Dentro do arquivo executável existe somente uma referência à necessidade de criação de 256 bytes no topo da seção `bss` (veja com `objdump -t`).

Um ponto importante a ser observado é o conflito dos registradores utilizados na chamada de procedimento. Na entrada do procedimento `main`, o registrador `%rdi` contém o valor de `argc` (primeiro parâmetro) enquanto o registrador `%rsi` contém o endereço do vetor de endereços `argv`. Porém, estes dois registradores são usados na chamada de procedimento `printf`. Para não perdê-los, usamos a convenção apresentada na Seção 3.3.2 onde o procedimento chamador salva os registradores:

- A linha 35 abre 0x80 (128 bytes) para salvamento de registradores. Este espaço comporta até 16 registradores, muito mais do que é necessário neste exemplo.
- Os comandos entre as linhas 39 e 41 salvam os registradores indicados. Dali para frente os parâmetros utilizados são os contidos nestes endereços e não nos registradores.

Agora vamos destacar como são mapeadas variáveis globais para a seção `BSS`. A forma de fazer isso varia de linguagem de programação para linguagem de programação. Na linguagem “C”, isto é obtido colocando a palavra reservada “`static`” na frente da declaração da variável global. Por exemplo:

```
...
static int alocaNaBSS;
...
```

Por motivos óbvios, a `BSS` não pode ter variáveis iniciadas. Por esta razão, a declaração

```
...
static int alocaNaDATA=10;
...
```

irá alocar a variável `alocaNaDATA` na seção `.data` e não na seção `.bss`.

É importante observar que a palavra reservada “`static`” também pode ser usada na frente de uma variável local, com semântica completamente diferente (veja Exercício 3.14).

Exercícios

- 5.1 O programa assembly apresentado nos Algoritmos 5.2 e 5.3 mapeia as variáveis em endereços de memória conforme descrito na Tabela 15. Reescreva o programa utilizando somente registradores (não precisam ser estes).

Variável	Registrador	End. Memória
\$?		-0x40(%rbp)
arge	%rdx	-0x38(%rbp)
argv	%rsi	-0x30(%rbp)
argc	%rdi	-0x28(%rbp)
bytes_escritos	%r13	-0x20(%rbp)
bytes_lidos	%r12	-0x18(%rbp)
fd_write	%r11	-0x10(%rbp)
fd_read	%r10	-0x08(%rbp)

Tabela 15 – Associação de variáveis e seus locais de armazenamento no programa apresentado nos Algoritmos 5.2 e 5.3

```

1  .section .data
2  str1:  .string "Copiando s em s\n"
3  str2:  .string "Erro ao abrir arquivo s\n"
4
5  # Constantes
6  .equ READ_SERVICE, 0
7  .equ WRITE_SERVICE, 1
8  .equ OPEN_SERVICE, 2
9  .equ CLOSE_SERVICE, 3
10 .equ EXIT_SERVICE, 60
11 .equ O_RDONLY, 0000
12 .equ O_CREAT, 0100
13 .equ O_WRONLY, 0001
14 .equ MODE, 0600
15
16 .section .bss
17 .equ TAM_BUFFER, 256
18 .lcomm BUFFER, TAM_BUFFER
19
20 # ----- Enderecos das variaveis na pilha:
21 # valor saida      :      : -0x40(%rsp)
22 # arge             : %rdx : -0x38(%rsp)
23 # argv            : %rsi : -0x30(%rsp)
24 # argc            : %rdi : -0x28(%rsp)
25 # bytes_escritos  : %r13 : -0x20(%rsp)
26 # bytes_lidos     : %r12 : -0x18(%rsp)
27 # fd_write        : %r11 : -0x10(%rsp)
28 # fd_read         : %r10 : -0x08(%rsp)
29
30 .section .text
31 .globl main
32 main:
33     pushq %rbp
34     movq  %rsp, %rbp
35     subq  $0x80, %rsp
36     movq  $0, -0x40(%rsp)
37
38 # Salva parametros argc, argv, arge
39     movq  %rdi, -0x28(%rbp)    # salva argc em -0x28(%rbp)
40     movq  %rsi, -0x30(%rbp)    # salva argv em -0x30(%rbp)
41     movq  %rdx, -0x38(%rbp)    # salva arge em -0x38(%rbp)
42
43 # Imprime mensagem de copia (str1)
44     movq  -0x30(%rbp), %rbx    # %rbx := argv
45     movq  8(%rbx), %rsi        # %rsi := argv[1]
46     movq  -0x30(%rbp), %rbx
47     movq  16(%rbx), %rdx       # %rdi := argv[2]
48     movq  $str1, %rdi
49     call  printf
50
51 # Cria fd para fd_read
52     movq  $OPEN_SERVICE, %rax
53     movq  -0x30(%rbp), %rbx
54     movq  8(%rbx), %rdi
55     movq  $O_RDONLY, %rsi
56     movq  $MODE, %rdx
57     syscall
58     movq  %rax, -0x08(%rsp)     # salva retorno da syscall em fd_read
59     cmpq  $0, %rax
60     jge  abre_argv_2
61     movq  -0x30(%rbp), %rbx    # deu erro: imprime msg e finaliza
62     movq  8(%rbx), %rsi
63     movq  $str2, %rdi
64     call  printf
65     movq  $-1, -0x40(%rsp)     # saida ($?) := -1
66     jmp  fim_pgma
67 .include "copiaS_parte2.s"

```

Algoritmo 5.2 – Programa copiaS.s: primeira parte da tradução do algoritmo 5.1

```

68 # Cria fd para fd_write
69 abre_argv_2:
70     movq $OPEN_SERVICE, %rax
71     movq -0x30(%rbp), %rbx
72     movq 16(%rbx), %rdi
73     movq $O_CREAT, %rsi
74     orq $O_WRONLY, %rsi
75     movq $MODE, %rdx
76     syscall
77     movq %rax, -0x10(%rsp)      # salva retorno da syscall em fd_write
78     cmpq $0, %rax
79     jge primeira_leitura
80     movq -0x30(%rbp), %rbx      # deu erro: imprime msg e finaliza
81     movq 16(%rbx), %rsi
82     movq $str2, %rdi
83     call printf
84     movq $-1, -0x40(%rsp)      # saida ($) := -1
85     jmp fecha_fd_read
86
87 primeira_leitura:
88     movq $READ_SERVICE, %rax
89     movq -0x08(%rsp), %rdi
90     movq $BUFFER, %rsi
91     movq $TAM_BUFFER, %rdx
92     syscall
93     movq %rax, -0x18(%rsp)
94
95 while:
96     cmpq $0, -0x18(%rsp)
97     jle fim_pgma
98
99 # escrita
100    movq $WRITE_SERVICE, %rax
101    movq -0x10(%rsp), %rdi
102    movq $BUFFER, %rsi
103    movq -0x18(%rsp), %rdx
104    syscall
105    movq %rax, -0x20(%rsp)
106
107 # nova leitura
108    movq $READ_SERVICE, %rax
109    movq -0x08(%rsp), %rdi
110    movq $BUFFER, %rsi
111    movq -0x20(%rsp), %rdx
112    syscall
113    movq %rax, -0x18(%rsp)
114    jmp while
115
116 fecha_fd_write:
117    movq $CLOSE_SERVICE, %rax
118    movq -0x10(%rsp), %rdi
119    syscall
120
121 fecha_fd_read:
122    movq $CLOSE_SERVICE, %rax
123    movq -0x08(%rsp), %rdi
124    syscall
125
126 fim_pgma:
127    movq -0x40(%rsp), %rdi
128    movq $EXIT_SERVICE, %rax
129    syscall

```

Algoritmo 5.3 – Segunda parte da tradução do programa copiaS.s

6. A seção Heap

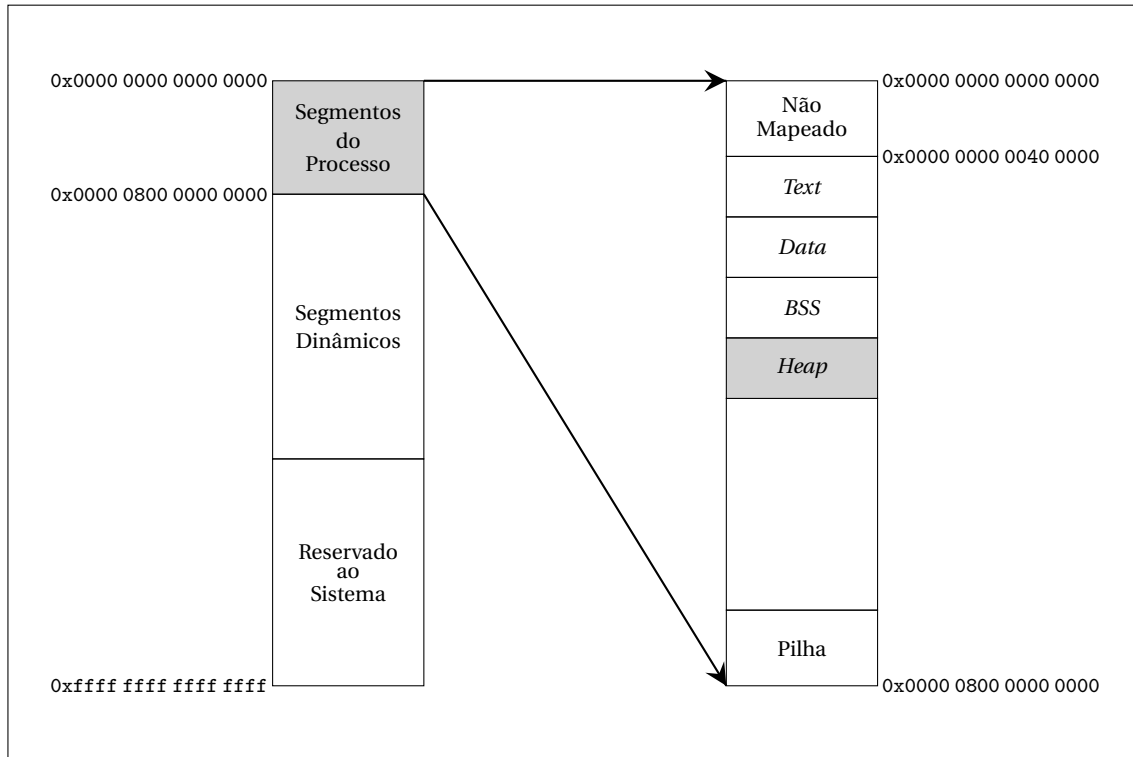


Figura 17 – Espaço Virtual de um Processo: *Heap*

As seções `.data`, `.bss` são usadas para armazenar estruturas de dados de tamanho fixo. A pilha também é destinada ao mesmo fim, apesar de poder armazenar várias instâncias de uma mesma variável.

Estas áreas são adequadas para armazenar variáveis e estruturas de dados estáticos como variáveis escalares, vetores e matrizes. Porém estes não são locais ideais para armazenar variáveis e estruturas de dados dinâmicas como listas, pilhas, filas, árvores, entre outras.

Como exemplo, considere o funcionamento de uma fila, que apresenta a seguinte API: `inicia()`, `finaliza()`, `insere()` e `retira()`.

Considere agora o funcionamento das operações `insere()` e `retira()`. Ao ser chamada, o procedimento `insere()` solicita um novo espaço de memória no qual irá armazenar dados enquanto a operação `retira()` devolve o espaço de memória.

Um mecanismo simples de implementar esta API é através de um vetor e um ponteiro para o último elemento. Aliás, todas as estruturas dinâmicas podem ser implementadas em vetores, mas esta implementação apresenta alguns problemas. O principal deles é que o tamanho do vetor é determinado em tempo de compilação (ou seja, é fixo).

A região da *heap* é um local criado para alocar estruturas dinâmicas que apresentam a vantagem de ter um tamanho variável. A Figura 6 mostra onde ela fica localizada. A seta indica que ela pode crescer “para baixo” em tempo de execução. O limite de crescimento é determinado pelo topo da seção *stack* (ou seja, pelo valor do registrador `%rsp`).

A palavra inglesa *heap* pode ser traduzida como “monte” ou “amontoado”. A ideia é que ela armazena um “amontoado” de memória que pode ser disponibilizado ao programa quando solicitado.

Este capítulo está organizado da seguinte forma: a Seção 6.1 descreve como a API do Posix trata a *heap* e algumas formas de implementá-la. A Seção 6.2 descreve o funcionamento dos procedimentos implementados na `libc` para gerenciar a *heap*. Por fim, a Seção 6.3 aborda alguns dos problemas no gerenciamento da *heap* assim como soluções propostas.

6.1 Gerenciamento da *heap*

A região de memória onde reside a *heap* é delimitada de um lado pela *bss* e do outro por uma variável chamada ‘*brk*’.

No sistema operacional há uma variável *brk* para cada processo, e a API do POSIX[7] tem uma chamada de sistema específica para alterar o valor desta variável reproduzida abaixo da Tabela 14:

%rax	Nome	%rdi	%rsi	%rdx	Comentário
12	<i>brk</i>	Novo valor da <i>brk</i> (Se zero, retorna valor atual de <i>brk</i> em %rax)			Altera a altura da <i>brk</i> , o que aumenta ou diminui a área da <i>heap</i> .

Na tabela de processos, cada entrada armazena informações sobre um processo. Uma destas informações é a altura atual da *heap* que é armazenada numa variável. Assim, cada processo tem exatamente uma variável que determina o limite da *heap* daquele processo.

A chamada ao sistema *brk* altera o valor desta variável como sugerido pelo Algoritmo 6.1. Neste algoritmo, a variável *brk_size* armazena o valor atual da altura da *heap*, *TabProcesso* representa a tabela de processos e *PID*¹ indica o identificador do processo que fez a chamada ao sistema.

```

1  Se (%rdi == 0)
2    %rax = TabProcesso[PID].brk_size;
3  Senao
4    TabProcesso[PID].brk_size = %rdi;
5  FimSe

```

Algoritmo 6.1 – Funcionamento da chamada *brk*

Até o momento, foi explicado que é possível alterar o “altura” da *heap*. É natural imaginar que cada *malloc()* implica na execução de uma *syscall*, mas isto causaria problemas. Por exemplo, considere as seguintes operações:

1. alocação de 100 bytes (*a=malloc(100)*)
2. alocação de 110 bytes (*b=malloc(110)*)
3. alocação de 120 bytes (*c=malloc(120)*)
4. alocação de 130 bytes (*d=malloc(130)*)
5. liberação do espaço de 120 bytes (*free(c)*)
6. alocação de 120 bytes (*e=malloc(120)*)

Onde será alocado a última operação? A primeira opção é após o espaço alocado para *d* e a segunda opção é alocá-la no espaço liberado de *c*.

Não é muito difícil deduzir que a melhor opção é a segunda (no espaço liberado de *c*), mas para tal é necessário saber a situação de cada bloco alocado (se livre ou ocupado), seu tamanho, etc. Em outras palavras, é necessário implementar um gerenciador do espaço na *heap*.

Antes de entrar em detalhes da implementação do gerenciador da *heap*, considere primeiro sua interface com o usuário (API). Um gerenciador minimalista teria uma API composta por quatro procedimentos:

void iniciaAlocador() cria as estruturas de dados para gerenciamento da *heap*.

void finalizaAlocador() libera as estruturas criadas por *iniciaAlocador()*.

void alocaMem(int num_bytes)* solicita a alocação de um bloco com *num_bytes* bytes na *heap* e que retorne o endereço inicial deste bloco.

int liberaMem(void bloco)* devolve para a *heap* o bloco que foi alocado por *alocaMem*. O parâmetro *bloco* é o endereço retornado por *alocaMem*.

Um exemplo de uso destes procedimentos é apresentado no Algoritmo 6.2.

¹ *Process ID*

```
1 void *a;
2 int main ( int argc, char **argv){
3     void *b;
4     iniciaAlocador();
5     a = alocaMem(100);
6     b = alocaMem(200);
7     strcpy(a, "Preenchimento de Vetor");
8     strcpy(b, a);
9     liberaMem (a);
10    liberaMem (b);
11    a = alocaMem(50);
12    liberaMem (a);
13    finalizaAlocador();
14 }
```

Algoritmo 6.2 – Exemplo de uso da API genérica de alocação de memória na *heap*

Este algoritmo tem duas variáveis do tipo endereço. A variável *a* declarada na linha 1 é global (seção *.data*) enquanto que a variável *b* declarada na linha 3 é local (seção *stack*).

A iniciação do gerenciador da *heap* ocorre antes de usar as funções de alocação e liberação, na linha 4. A finalização ocorre na linha 13, depois do que as funções de alocação e liberação não podem mais ser usadas.

As linhas 5 e 6 solicitam que o alocador disponibilize blocos contíguos na *heap* com 100 e 200 bytes respectivamente. O endereço do bloco retornado pela linha 5 é armazenado na variável *a* e o retornado pela linha 6 é armazenado na variável *b*.

As linhas 9 e 12 avisam ao gerenciador que o espaço alocado para o endereço apontado por *a* está disponível para futuras alocações.

A linha 11 faz uma nova solicitação de alocação, desta vez de um bloco de 50 bytes. Dependendo da forma com que o gerenciador da *heap* for desenvolvido, o endereço retornado por esta nova alocação pode ser o mesmo retornado pela linha 5, reaproveitando um bloco livre.

A implementação do gerenciador da *heap* é o objeto das próximas seções. A Seção 6.1.1 apresenta uma abordagem ingênua onde cada pedido de alocação será sempre respondido com um novo espaço na *heap*. A Seção 6.1.2 apresenta uma segunda abordagem que organiza os blocos em uma lista encadeada que permite realocação, mas com custo computacional elevado na busca por blocos livres.

6.1.1 Primeira tentativa

Esta primeira tentativa de um gerenciador é uma abordagem ingênua. A ideia é abrir mais espaço na *heap* a cada novo pedido sem reaproveitar os blocos liberados. As funções da API teriam a seguinte funcionalidade:

`void iniciaAlocador():` executa syscall `brk` para obter o endereço do topo corrente da *heap* e o armazena em uma variável global, `topoInicialHeap`;

`void finalizaAlocador():` executa syscall `brk` para restaurar o valor original da *heap* contido em `topoInicialHeap`;

`void* alocaMem(int num_bytes):` executa syscall `brk` para abrir um bloco de `num_bytes`, retornando o endereço inicial deste bloco;

`int liberaMem(void* bloco):` não faz nada.

Esta abordagem é ingênua porque considera que o espaço na *heap* é infinito ao nunca reutilizar os blocos liberados pelo programa.

O programa apresentado no Algoritmo 6.3 é um exemplo de problema gerado por esta abordagem. Ele apresenta um laço que aloca um bloco e em seguida o libera. Se o gerenciador souber reutilizar o espaço na *heap*, o programa irá utilizar sempre o mesmo bloco e os endereços impressos serão sempre o mesmo. Porém, na abordagem aqui descrita, o programa provavelmente será abortado por invadir a área da *stack*.

6.1.2 Segunda tentativa

A primeira tentativa não foi projetada para reusar o espaço que foi alocado e posteriormente liberado. Para poder reutilizar um bloco já liberado, é necessário manter informações sobre cada bloco, em especial tamanho e se o bloco está livre ou ocupado.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  int main (int argc, char** argv)
5  {
6      void *a;
7      int i;
8
9      for (i=0; i<100; i++) {
10         a=malloc(100);
11         strcpy(a, "TESTE");
12         printf ("%p %s\n", a, (char*) a);
13         free(a);
14     }
15
16     return (0);
17 }

```

Algoritmo 6.3 – Programa que imprime os endereços de alocações sucessivas

Várias abordagens são possíveis, e elas normalmente dividem a região da *heap* em duas partes: aquela que armazena informações gerenciais sobre cada bloco (livre ou ocupado, tamanho, etc.) e aquela que armazena o bloco (cujo início é o endereço retornado pelo procedimento `alocaMem()`).

A funcionalidade das funções genéricas é a seguinte:

`void iniciaAlocador()` Executa syscall `brk` para obter o endereço do topo corrente da *heap* e o armazena em uma variável global, `topoInicialHeap`.

`void finalizaAlocador()` Executa syscall `brk` para restaurar o valor original da *heap* contido em `topoInicialHeap`.

`int liberaMem(void* bloco)` indica que o bloco está livre.

`void* alocaMem(int num_bytes)`

1. Procura um bloco livre com tamanho maior ou igual à `num_bytes`.
2. Se encontrar, indica que o bloco está ocupado e retorna o endereço inicial do bloco;
3. Se não encontrar, abre espaço para um novo bloco usando a syscall `brk`, indica que o bloco está ocupado e retorna o endereço inicial do bloco.

A questão que não foi abordada é o mecanismo de buscar o próximo bloco livre ou melhor, qual estrutura de dados usar.

Jonathan Bartlett[3] apresenta uma alternativa bastante simples para fins ilustrativos², porém ineficiente.

A abordagem implementada por Bartlett utiliza uma lista ligada onde o começo e o fim da lista são variáveis globais que aqui chamaremos de `início_heap` e `topo_heap`.

Cada nó da lista contém três campos:

- o primeiro indica se o bloco está livre (igual a 0) ou se está ocupado (igual a 1);
- o segundo indica o tamanho do bloco. Também é usado para determinar o endereço do próximo nó da lista.
- o terceiro é o bloco alocado (o primeiro endereço é retornado por `alocaMem()`).

O funcionamento do algoritmo é apresentado na Figura 18 para a seguinte sequência de comandos:

(1) `void iniciaAlocador():`

- Faz uma chamada ao sistema com o serviço `brk` para localizar o endereço do topo da *heap* e o armazena na variável que indica o topo da *heap*.
- Inicia a variável que indica o início da *heap* apontando para o topo da *heap*. Isto indica que a *heap* está vazia;

(2) `x = alocaMem(100):`

² o livro de Bartlett também apresenta o código assembly completo (para x86-32)

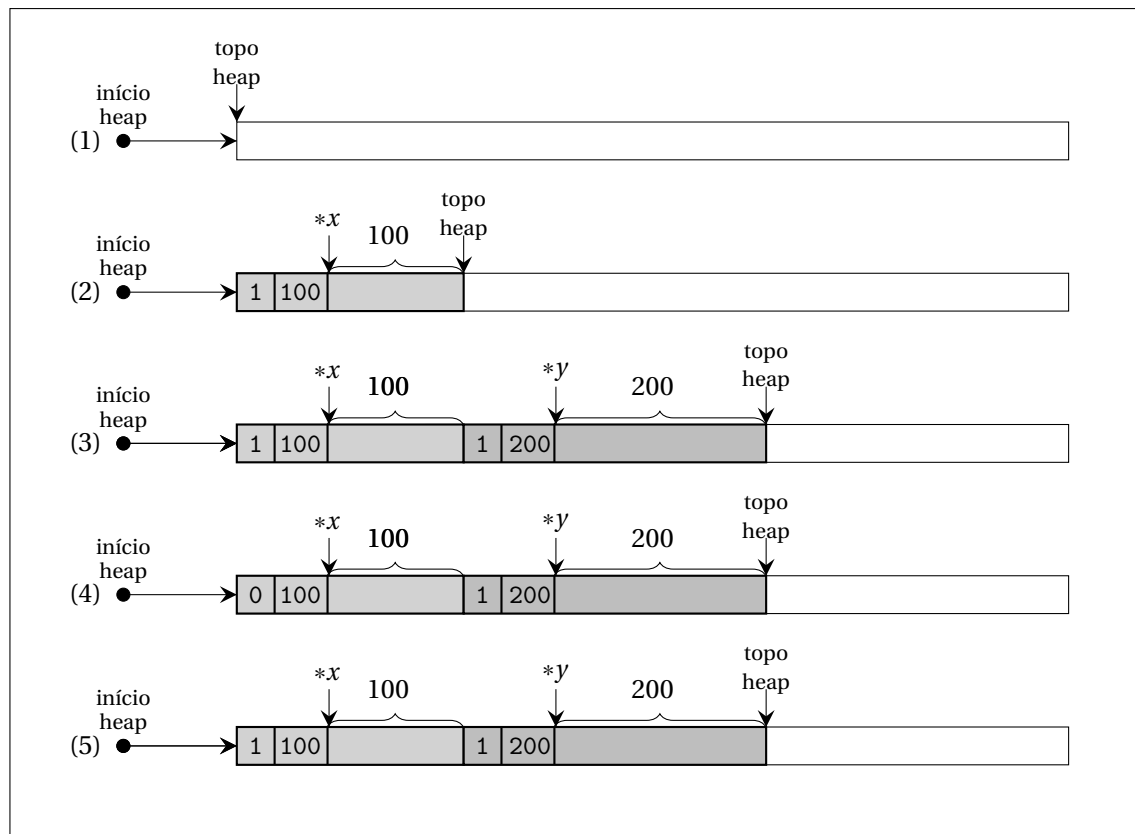


Figura 18 – Alocador de Bartlett

- Como a lista está vazia, cria um novo nó usando o serviço `brk` para aumentar o topo da *heap* em 100 + 16 bytes. Os 16 bytes correspondem às informações gerenciais. Este espaço corresponde ao novo nó.
- No campo livre/ocupado indica ocupado;
- No campo tamanho armazena o tamanho do bloco (100);

(3) `y = alocaMem(200);`

- Percorre a lista procurando por um nó livre. Não encontra e por isso cria um novo nó usando o serviço `brk` para aumentar o topo da *heap* em 200 + 16 bytes. Os 16 bytes correspondem às informações gerenciais. Este espaço corresponde ao novo nó.
- No campo livre/ocupado indica ocupado;
- No campo tamanho armazena o tamanho do bloco (200);

(4) `int liberaMem(x)`: O parâmetro passado (*x*) aponta para o primeiro byte do bloco. Subtrai 16 e encontra o campo livre/ocupado. Armazena 0 (zero=livre) neste campo.

(5) `x = alocaMem(100)`: Percorre a lista procurando por um nó livre com bloco maior ou igual a 100 bytes. Encontra o primeiro nó e muda o campo livre/ocupado para ocupado. Retorna o endereço do primeiro byte do bloco.

6.1.3 Discussão

O algoritmo proposto na seção anterior tem a virtude de ser simples, mas apresenta alguns problemas graves:

1. a complexidade para alocar um novo nó é $O(n)$, onde n é o número de elementos da lista;
2. se os nós livres puderem ser divididos quando o tamanho for menor que o tamanho do bloco, podem ser criados vários nós de tamanho muito pequeno. Se forem muitos nós pequenos, podem ser criados muitos “buracos” na *heap*, o que é chamado fragmentação.

3. não há prevenção de *buffer overflow*. Se ocorrer, irá bagunçar as informações gerenciais provavelmente abortando o programa na próxima operação.

Porém, não é difícil imaginar melhorias ao algoritmo:

- criar duas listas ligadas, uma contendo somente os nós livres e outra contendo somente os nós ocupados.
- criar uma área somente para armazenar as informações gerenciais e outra somente para os blocos.
- minimizar a quantidade de vezes que é feita a chamada ao serviço `brk`. Uma solução é usar esta chamada para abrir um grande espaço na *heap* e preenchê-lo sob demanda. Quando este espaço estiver cheio, abre-se um novo e assim por diante.

6.2 Funções de gerenciamento da libc

A `libc` apresenta quatro funções para alocar e liberar blocos de memória na *heap*:

`void *malloc (size_t)` funcionalidade semelhante ao `alocaMem` descrito na Seção 6.1.1;

`void free(void *ptr)` funcionalidade semelhante ao `liberaMem` descrito na Seção 6.1.1;

`void *calloc(size_t nmemb, size_t size)` aloca memória para um vetor de `nmemb` elementos de tamanho `size`

`void *realloc(void *ptr, size_t size)` permite mudar o tamanho de um bloco.

Originalmente o gerenciador desta família de funções foi desenvolvida por Doug Lea (chamada `dlmalloc` - Doug Lea Malloc) e a versão atualmente usada, `ptmalloc2`, é baseada no primeiro³.

Os algoritmos acima são complicados e não serão explicados aqui. Para os interessados, sugerimos as seguintes fontes:

- Knuth [21] talvez tenha sido o primeiro a expor o problema em um livro. Ele apresenta o problema, o que se espera maximizar e minimizar e propõe algoritmos que vão desde listas encadeadas (como implementado por Bartlett) até o *Buddy Algorithm*. Discute também as opções de escolha de nós (*first fit*, *best fit* ou *next fit* (veja Exercício 6.2e). Para mais informações consulte a seção 2.5 Dynamic Storage Allocation.
- Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles [32] apresentam um levantamento mais detalhado de técnicas de gerenciamento da *heap*.
- Doug Lea mantém um sítio descrevendo o algoritmo `dlmalloc` em detalhes⁴.

6.3 Problemas decorrentes

Alocar espaço na *heap* apresenta muitas vantagens, porém envolve alguns problemas.

Um dos problemas é conhecido por *memory leak* (algo como “vazamento de memória”), e ocorre quando um programa “perde” a referência aos endereços de memória alocados. Um exemplo de programa com este problema está apresentado no Algoritmo 6.4.

O Algoritmo 6.4 e o Algoritmo 6.3 são parecidos. A diferença entre eles está na localização do comando `free` (a). O Algoritmo 6.4 libera o espaço alocado uma única vez, um erro muito comum entre programadores iniciantes. As referências a todas as outras 99 alocações ficarão ocupando espaço, mas não serão liberadas.

O *memory leak* é um problema que aparece por descuido do programador, e a maneira mais simples de evitá-lo é utilizando o que se chama de programação defensiva. Seacord [40] aborda este e outros problemas (por exemplo, na *stack*) descrevendo como se escrever programas robustos.

A detecção e remoção de *memory leak* é chamada *garbage collection*. Várias abordagens foram apresentadas e sugerimos as seguintes referências para mais informações:

- Wilson *et alli* [32] avaliam o custo de uma classe de *garbage collectors*;
- Jones *et alli* [37] descrevem detalhadamente os *garbage collectors*;

³ <http://www.eecs.harvard.edu/mdw/course/cs61/mediawiki/images/5/51/Malloc3.pdf>

⁴ <http://g.oswego.edu/dl/html/malloc.html>

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  int main (int argc, char** argv)
5  {
6      void *a;
7      int i;
8
9      for (i=0; i<100; i++) {
10         a=malloc(100);
11         strcpy(a,"TESTE");
12         printf ("%p %s\n", a, (char*) a);
13     }
14     free(a);
15
16     return (0);
17 }

```

Algoritmo 6.4 – Programa com *memory leak*

Exercícios

- 6.1 A região da *stack* e da *heap* podem ser alterados em tempo de execução. Se o valor de `brk` for maior ou igual ao valor de `%rsp`, a execução deve ser abortada. Pensando no modelo de paginação, descreva um algoritmo que avisa quando isto ocorrer.
- 6.2 No algoritmo sugerido na Figura 18, pergunta-se:
- a) se o último `alocaMem` solicitar 200 bytes (ao invés de 100), qual o resultado?
 - b) se o último `alocaMem` solicitar 50 bytes (ao invés de 100), o nó livre poderia ser dividido em dois. O primeiro seria usado para comportar os 50 bytes e o segundo seria livre. Qual o espaço disponível para o segundo bloco?
 - c) O algoritmo permite a existência de dois nós livres consecutivos. Indique como proceder para fundir dois (ou três) nós livres consecutivos. Seja x o tamanho do primeiro bloco livre e y o tamanho do segundo. Qual o tamanho do bloco livre após a fusão?
 - d) Quais vantagens e desvantagens de utilizar uma lista duplamente encadeada. Ela é útil na fusão de nós livres?
 - e) Por que o algoritmo *Buddy* é interessante para ser usado no gerenciamento da *heap*?
 - f) A Seção 6.1.3 levanta o problema da fragmentação. Projete um algoritmo que usa os procedimentos `alocaMem` e `liberaMem` que lida com blocos com mais de 100 bytes e que deixe pelo menos 1 milhão de fragmentos livres de 10 bytes cada.
- 6.3 O programa apresentado no Algoritmo 6.5 causa *overflow* dependendo do tamanho de `argv[1]`.
- Em sua opinião, o programa vai abortar ou vai finalizar normalmente? Justifique.
 - Se ele abortar, indique em qual linha.
 - E se mudar o `TAMANHO` para 10?
 - Implemente e comprove.
- 6.4 Desenhe o mapa da *heap* após a execução do Algoritmo 6.6.
- 6.5 Uma aluna observou que o procedimento `sbrk()`⁵ tem funcionalidade equivalente à chamada ao sistema `brk` descrita neste capítulo. Porém, ao executar o código indicado no Algoritmo 6.7 obteve o seguinte resultado:

```

> ./algoEstranho
0x1f55000
0x1f76000

```

⁵ `man 2 sbrk`

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  #define TAMANHO 1024
6
7  int main(int argc, char **argv) {
8      char *p, *q;
9      int i;
10
11     p = malloc(TAMANHO);
12     q = malloc(TAMANHO);
13     if (argc >= 2)
14         strcpy(p, argv[1]);
15
16     free(q);
17     free(p);
18
19     return 0;
20 }

```

Algoritmo 6.5 – Overflow na *heap*

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  int main(int argc, char **argv) {
6      char *p, *q, *t;
7      int i;
8      p = malloc(1024);
9      q = malloc(1024);
10     for (i=0; i<100; i++){
11         free (p);
12         p = malloc(1);
13         p = malloc(1024);
14         t=p; p=q; q=t;
15     }
16     free(q);
17     free(p);
18     return 0;
19 }

```

Algoritmo 6.6 – Fragmentação

```

> ./algoEstranho
0xcc0000
0xce1000
> ./algoEstranho
0x17b3000
0x17d4000
> ./algoEstranho
0x228d000
0x22ae000

```

- Porque os valores de `i1` e `i2` estão sempre distantes um número igual de bytes?
- Porque os valores de `i1` e `i2` nunca se repetem?
- Dica: troque as linhas 7 e 8 e veja o que ocorre.


```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main(int argc, char **argv) {
5     void *i1, *i2;
6
7     i1=sbrk(0);
8     printf("%p\n",i1);
9     i2=sbrk(0);
10    printf("%p\n",i2);
11    return (0);
12 }
```

Algoritmo 6.7 – Algo estranho

Projetos de Implementação

6.1 implemente o algoritmo proposto na Seção 6.1.2 em assembly.

6.2 implemente as seguintes variações:

- a) faça a fusão de nós livres;
- b) use duas regiões: uma para as informações gerenciais e uma para os nós;
- c) minimize o número de chamadas ao serviço `brk` alocando espaços múltiplos de 4096 bytes por vez. Se for solicitado um espaço maior, digamos 5000 bytes, então será alocado um espaço de $4096 * 2 = 8192$ bytes para acomodá-lo.
- d) utilize duas listas: uma com os nós livres e uma com os ocupados;
- e) escreva variações do algoritmo de escolha dos nós livres:
 - *first fit*: percorre a lista do início e escolhe o primeiro nó com tamanho maior ou igual ao solicitado;
 - *best fit*: percorre toda a lista e seleciona o nó com menor bloco, que é maior do que o solicitado;
 - *next fit*: é *first fit* em uma lista circular. A busca começa onde a última parou.

6.3 implemente (em C) um gerenciador que usa o algoritmo *Buddy*.

6.4 implemente um procedimento que imprime um mapa da memória da região da `heap` em todos os algoritmos propostos aqui. Cada byte da parte gerencial do nó deve ser impresso com o caractere "#". O caractere usado para a impressão dos bytes do bloco de cada nó depende se o bloco estiver livre ou ocupado. Se estiver livre, imprime o caractere "-". Se estiver ocupado, imprime o caractere "+".

PARTE II

APLICATIVOS BÁSICOS

7. Aplicativos Básicos

Este texto usa os termos “Aplicativo Básico” para designar qualquer arquivo associado à execução de programas. Os formatos e os aplicativos que serão descritos aqui são apresentados esquematicamente na Figura 19.

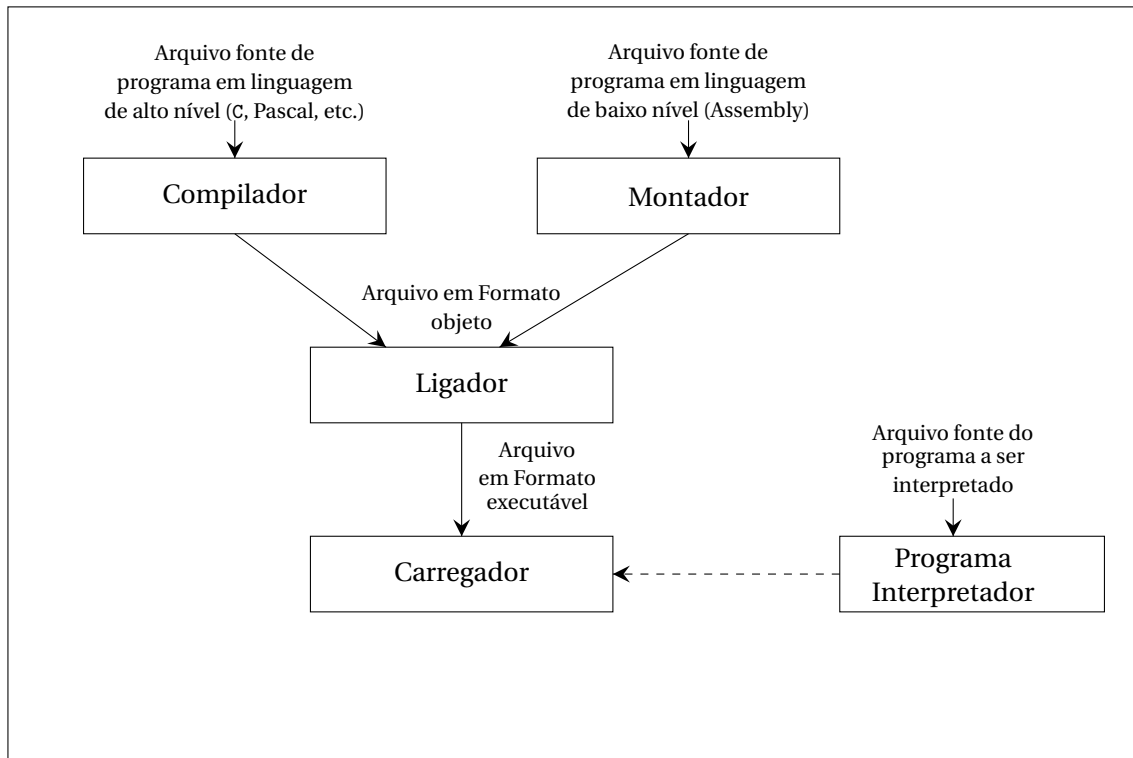


Figura 19 – Conversão de Arquivos

A figura mostra os seguintes formatos de arquivos que podem ser vistos como executáveis:

- **Arquivos fonte em linguagem de alto nível** como C e Pascal (para citar só duas). Os arquivos deste grupo estão sempre em formato texto (ASCII, UTF-8, etc.), e apesar dos programadores imaginarem a execução destes arquivos, a máquina não os executa diretamente. Para tal, é necessário convertê-los para arquivos em formato objeto e depois em arquivo em formato executável que é o formato que a máquina consegue executar.
- **Arquivos fonte em linguagem de baixo nível** como assembly. Assim como o grupo anterior, os arquivos deste grupo estão sempre em formato texto (ASCII, UTF-8, etc.). Eles não passam pelo compilador, mas sim pelo montador para gerar arquivos objeto. A diferença é que aqui uma instrução no arquivo corresponde a normalmente uma instrução em linguagem de máquina.
- **Arquivos objeto estão em um formato intermediário.** Este grupo de arquivos são organizados em um formato especificado pelo sistema operacional. No caso do linux é o formato ELF objeto.
- **Arquivos executáveis** estão em um formato pronto para execução pela máquina. Assim como o formato objeto, este grupo de arquivos são organizados em um formato especificado pelo sistema operacional. No caso do linux é o formato ELF executável.
- **Arquivos fonte do programa em formato texto a ser interpretado.** O programa interpretador é um programa em formato executável. Ele recebe o arquivo fonte como entrada e o executa (ele não o transforma em outro formato de arquivo).

Os arquivos fonte são normalmente codificados como texto (ASCII, UTF-8, etc.), pois são normalmente gerados por humanos para serem compreendidos por humanos¹. Já os arquivos objeto e executável não se restringem só àqueles caracteres (que aparecem de vez em quando), e por isso são chamados de arquivos binários. Aliás, isto explica porque alguns dos aplicativos utilizados neste livro (como `readelf`, `as`, `objdump` e `ld`) estão agrupados no pacote `binutils`².

Observe a caixa referente ao programa ligador. Ele recebe como entrada arquivos no formato binário objeto e gera como saída arquivos binários em formato executável. Porém, o interessante é que os arquivos de entrada podem ser originários tanto de programas de alto nível quanto de baixo nível, sugerindo que é possível que um programa executável seja composto por pedaços escritos em assembly e pedaços escritos em linguagem de alto nível. Sugere também que é possível usar várias linguagens de alto nível diferentes, gerar os respectivos arquivos objetos e juntá-los em um executável. Isto realmente é possível, como será visto no Capítulo 9.

Ainda analisando a Figura 19, vamos agora descrever os aplicativos envolvidos, os aplicativos básicos.

Compiladores: Compiladores são programas que convertem arquivos que contém programas escritos em linguagens de alto nível (formato texto) para arquivos em um formato intermediário denominado arquivo objeto. Exemplos de linguagens de alto nível incluem “C”, “Java”, “Fortran”, “Pascal”, entre outros.

Montadores: Montadores são programas que convertem arquivos que contém programas em linguagem assembly (formato texto) para o formato de código objeto.

Ligadores: são programas que convertem arquivos em formato objeto para formato executável.

Carregadores: Carregam os arquivos que contém formato executável para a execução. Para isto ele lê partes dos arquivos objeto e copia estas partes para a memória.

Interpretadores: são programas executáveis que recebem como entrada arquivos contendo uma linguagem conhecida pelo interpretador como por exemplo, Bytecode, Ruby, PHP. O interpretador comporta-se como uma máquina de execução específica para aquela linguagem.

O restante desta seção detalha o funcionamento, a forma de se utilizar e a funcionalidade destas ferramentas.

Compilador, montador e ligador são programas que podem ser invocados diretamente pelo programador.

Como exemplo destes programas e do efeito que eles produzem, digite e salve o Algoritmo 7.1 abaixo em um arquivo, por exemplo, `esqueletoC.c`.

```
1 int main (long int argc, char **argv)
2 {
3     return 13;
4 }
```

Algoritmo 7.1 – Arquivo `esqueletoC.c`

O compilador recebe este arquivo como entrada e gera um arquivo objeto (`esqueletoC.o`). Por fim, use o arquivo objeto para gerar o executável como descrito abaixo.

```
> gcc -c esqueletoC.c -o esqueletoC.o
> gcc esqueletoC.o -o esqueletoC
> ./esqueletoC
> echo \ $?
13
```

Os arquivos `esqueletoC.c`, `esqueletoC.o` e `esqueletoC` são respectivamente os programas fonte, objeto e executável sendo que cada um está armazenado em um formato específico.

O arquivo `esqueletoC.c` (programa fonte) segue o formato especificado da linguagem C.

O arquivo `esqueletoC.o` e o arquivo `esqueletoC` estão em um formato que não usa somente caracteres ASCII, e por isso é ilegível para a maior parte da raça humana. Porém, existem programas que imprimem o conteúdo deste tipo de arquivo para uma linguagem que utiliza caracteres ASCII (o que não significa dizer que os tornam legíveis). Um exemplo é o programa `objdump`, que pode ser usado em arquivos ELF objeto e executáveis. Exemplo:

¹ sim, eu sei que isto não é 100% verdadeiro, pois já encontrei alguns contraexemplos.

² <http://www.gnu.org/software/binutils/>

```
> gcc -c esqueletoC.c -o esqueletoC.o
> objdump -s esqueletoC.o

esqueletoC.o:      file format elf64-x86-64

Contents of section .text:
 0000 554889e5 897dfc48 8975f0b8 0d000000  UH...}.H.u.....
 0010 5dc3                                ].
Contents of section .comment:
 0000 00474343 3a202855 62756e74 7520342e  .GCC: (Ubuntu 4.
 0010 382e342d 32756275 6e747531 7e31342e  8.4-2ubuntu1~14.
 0020 30342920 342e382e 3400          04) 4.8.4.
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  ....zR..x..
 0010 1b0c0708 90010000 1c000000 1c000000  ....
 0020 00000000 12000000 00410e10 8602430d  ....A....C.
 0030 064d0c07 08000000          .M.....
```

Como foi dito antes, a impressão não é exatamente o que se pode chamar de “legível”.

Um outro aspecto importante é que o formato do arquivo objeto e executável é definido pelo Sistema Operacional. No linux, este formato é o ELF (Executable and Linking Format), que foi originariamente desenvolvido e publicado pelo UNIX System Laboratories (USL) como parte da *Application Binary Interface (ABI)*.

Já para o Sistema Operacional Windows foram projetados vários modelos de execução. Como exemplos, temos o formato COM, EXE e mais recente o PE (*Portable Executable File Format*).

Como quem define o formato do arquivo executável é o Sistema Operacional, programas executáveis gerados em um sistema operacional não pode ser executado diretamente em outro sistema operacional (mesmo que ambas as máquinas usem o mesmo processador).

Quando tal se faz necessário, é comum usar interpretadores ou emuladores do modelo executável em questão. Uma forma bastante esperta de se fazer isso é ler o o arquivo em formato executável “A” e reorganizá-lo no formato executável “B” para só então colocá-lo em execução. É assim que o wine³ trabalha.

Este texto irá detalhar alguns aspectos do formato de arquivos ELF. Sugerimos ao leitor que digite `objdump -s esqueletoC` para ver as seções em que o arquivo `esqueletoC` está dividido (section `.init`, `.plt`, `.text`, `.fini`). Destas, destacamos a seção `.text`, que contém os comandos em linguagem de máquina para o referido programa.

Uma vez apresentado o formato objeto e o formato executável (que é gerado pelo ligador), vamos abordar o carregador. O carregador é uma parte do Sistema Operacional que é capaz de ler um arquivo executável e colocá-lo na memória para execução. Basicamente ele executa as seguintes tarefas:

1. identifica as seções de um arquivo executável;
2. solicita uma nova entrada na tabela de processos do SO;
3. aloca espaço de memória virtual para o programa;
4. coloca o processo na fila de execução.

Ao final desta sequência, o Sistema Operacional terá um novo processo na tabela de processos, e o escalonará para execução de acordo com as regras de escalonamento de processos.

É comum encontrar dois tipos de carregadores: os carregadores absolutos e os relocáveis. Os carregadores absolutos alocam sempre a mesma porção de memória para um mesmo processo.

Isso parece estranho, e um exemplo pode ajudar. Todos os programas executáveis têm comandos assembly codificados em binário. Um dos comandos assembly que permitem o desvio do fluxo de execução é o comando do tipo `jump ENDEREÇO`. Quando se usa carregadores absolutos, o compilador assume que o programa começa em um determinado endereço da memória física e assim o `ENDEREÇO` contido no `jump` corresponde a um endereço físico de memória conhecido ANTES de o programa ser colocado em execução.

Isto implica dizer que o programa tem que ser colocado sempre no mesmo local da memória física para funcionar corretamente. Se esta região de memória estiver sendo ocupada, o Sistema Operacional adota uma das seguintes medidas:

1. retira o outro processo da memória (copia o outro processo para uma área em disco, chamada “memória swap”) e assim libera a área desejada para o novo processo;

³ <http://www.winehq.com/>

2. impede a execução do novo processo.

Como é de se imaginar, a primeira solução é a mais comum.

O segundo tipo de carregador é o **carregador relocável**. Neste caso, o programa pode ser colocado em qualquer local da memória para execução. O programa executável relocável é semelhante ao programa executável absoluto, exceto que os endereços de memória que ele referencia são ajustados quando ele é colocado para execução (na verdade, este ajuste pode ocorrer tanto em tempo de carregamento quanto em tempo de execução). Estes endereços ajustáveis são chamados *endereços relocáveis*. Esta solução é tão comum que muitos processadores incluem facilidades para efetuar a relocação em tempo de execução.

Os programas DOS com extensão “.exe” são exemplos de programas executáveis relocáveis, enquanto que os que tem extensão “.com” são exemplos de programas executáveis que usam relocador absoluto.

Apesar de mais flexíveis, os carregadores relocáveis também apresentam problemas:

- Os programas a serem executados não podem ser maiores do que a memória disponível. Ou seja, se o computador tiver 128M de memória RAM, ele não poderá executar programas que ocupem mais de 128M.
- Se a memória estiver ocupada por outros programas, (ou seja, há memória mas não está livre), estes programas terão de ser descarregados antes da execução do novo programa.

Várias abordagens foram tentadas para minimizar os problemas de memória citados acima, e as soluções propostas nem sempre são convincentes⁴.

É importante destacar que um dos maiores problemas para executar um processo⁵ é alocar espaço para ele na memória física. Cada sistema operacional tem seus próprios mecanismos mas estes mecanismos não serão detalhados aqui, pois são um tema longo normalmente abordado em livros de Sistemas Operacionais. Porém, abaixo é apresentada uma análise superficial das abordagens comuns:

Primeira Solução jogar o problema para o usuário: exigir que o computador tenha muita memória. Apesar de parecer pouco viável comercialmente, esta solução era muito comum, e para isto basta ver a “configuração mínima” exigida por certos programas⁶. Apesar de simples, ela pode comprometer as vendas dos produtos, principalmente se o programa precisar de uma grande quantidade de memória. Não é necessário dizer que esta solução não deixa os usuários felizes.

Segunda Solução jogar o problema para os programadores. Aqui, várias opções foram tentadas, mas a que é mais viável divide o programa em várias partes logicamente relacionadas. Como exemplo, considere os programas de cadastro que normalmente começam com uma tela que solicita que o usuário escolha por um opção entre *inclusão*, *alteração* e *exclusão*. As três opções são, normalmente, implementadas em trechos diferentes de código, e não precisam estar simultaneamente na memória. Neste tipo de aplicação, o programador indica quais são os três módulos e que eles devem ser carregados de forma excludente. Esta solução também está longe de ser a ideal, mas era encontrada frequentemente em programas da década de 70-80, quando os computadores tinham pouco espaço de memória (eu vivi esta realidade trabalhando com computadores com 16K de memória RAM). Esta abordagem remonta ao conceito de memória virtual denominada **overlay** (veja Seção D.1 do Apêndice D). Esta ideia é semelhante ao modelo de bibliotecas carregadas dinamicamente, onde o programador cria um programa principal (que pode ficar o tempo todo na memória) e as bibliotecas (sub-rotinas chamadas somente quando necessário). Em tempo de execução, o programa indica quais bibliotecas devem ser carregadas na memória, e o carregador (por vezes chamado de carregador-ligador) traz a biblioteca requisitada para a memória. Após o uso, o programador pode dizer ao Sistema Operacional que aquela biblioteca pode ser descarregada da memória.

Terceira Solução assumir que o sistema operacional é que é o responsável pelo gerenciamento da memória. Esta abordagem é a adotada pelos sistemas operacionais baseados em Unix como o linux onde o sistema operacional gerencia a memória por meio de um mecanismo chamado “paginação”.

A ideia básica da paginação é que um programa não acessa diretamente os endereços reais (na memória física), mas sim endereços virtuais. Trechos dos programas em memória virtual são então copiados para a memória real, e, cada vez que um endereço é acessado, o SO primeiro faz a conversão de endereços para descobrir a qual endereço

⁴ Como aconteceu em uma estória de Asterix onde o druida Amnesix “curou” uma pessoa que pensava ser um Javali. Ele o ensinou a ficar de pé, e assim notava-se menos o problema (veja história completa em *Asterix e o Combate dos Chefes*).

⁵ Na taxonomia de sistemas operacionais, um processo é um programa em execução.

⁶ Isto não quer dizer que todos os programas que apresentam uma “configuração mínima” recaem neste caso. Por vezes, ela indica o tamanho mínimo de memória necessário para partes básicas (aquelas que tem de ficar na memória ao longo de toda a execução do programa).

físico corresponde cada endereço virtual. Este processo é lento, e por isso grande parte das CPUs incorporaram este mecanismo para ganhar desempenho (veja Seção D.2 do Apêndice D).

Este assunto é muito denso, e é difícil determinar a melhor abordagem didática. A opção adotada aqui baseia-se na minha experiência em sala de aula, onde ficou claro que os alunos absorvem melhor o conteúdo quando aborda-se primeiro o funcionamento do carregador e do ligador estático (que servem como base para todo o processo) e só depois explicar o ligador dinâmico em conjunto com o ligador em tempo de execução⁷ (que usam conceitos adicionais sobre a base já explicada). Sendo assim, os próximos capítulos estão organizados como segue:

Para uma introdução um pouco mais aprofundada ao tema, o Capítulo 8 descreve sucintamente os formatos dos arquivos indicados na Figura 19.

Os capítulos seguintes são mais específicos. O Capítulo 9 apresenta o carregador estático e o ligador estático. Este capítulo é mais longo por apresentar o jargão técnico utilizado nos demais capítulos. Os capítulos seguintes só devem ser lidos após este capítulo for bem compreendido.

O Capítulo 10 aborda o ligador dinâmico e o ligador em tempo de execução⁸ (que complementa o trabalho do carregador) que trabalha com objetos compartilhados e dinâmicos. Por fim, o Capítulo 11 descreve o funcionamento de interpretadores.

⁷ *runtime linker*

⁸ *runtime linker*

8. Formatos de Programa

Este capítulo descreve o formato e organização dos arquivos apresentados na Figura 19. Os programas que os convertem são analisados nos próximos capítulos.

O restante desta seção está organizado como segue: a Seção 8.1 descreve os formatos de mais baixo nível, que incluem código de máquina, e apresenta o formato ELF. A Seção 8.1.1 apresenta o formato ELF objeto enquanto que a Seção 8.1.2 apresenta o formato ELF executável. Finalizando o capítulo, a Seção 8.1.3 descreve os formatos usados pelo linux para criar agregados de arquivos objetos: as bibliotecas.

8.1 Arquivos em formato Objeto e Executável

Nos primeiros computadores somente um processo por vez era colocado em execução. Por esta razão os arquivos executáveis eram mapeados diretamente em memória, e normalmente um arquivo executável era a imagem exata do estado que o programa deveria estar no início da execução.

Esta representação simples de arquivos era apropriada para os computadores da época, porém com o aumento da velocidade dos computadores, foram criados diversos mecanismos para otimização do uso dos recursos do sistema. Um dos mecanismos mais importantes foi otimizar o uso do processador e da memória permitindo que vários processos pudessem ser colocados em execução “simultânea”. Como resultado, foram criados formatos de arquivos executáveis mais apropriados a esta realidade.

Posteriormente novas demandas para racionalizar o uso dos recursos do sistema foram desenvolvidos para, por exemplo, compartilhar recursos entre os processos. Ao longo do tempo, novas demandas surgiram e os formatos de execução foram sendo adaptados continuamente para atingir estes (e novos) objetivos.

Nos sistemas operacionais UNIX, esta evolução se deu a partir de três formatos de arquivos. O primeiro formato de arquivos foi chamado de `a.out`¹ criado em 1968 [38], posteriormente substituído pelo formato COFF [14] e finalmente pelo atual ELF [8].

Esta seção concentra-se no formato ELF para arquivos executáveis e arquivos objeto, porém apresentamos a seguir algumas informações sobre os outros dois formatos:

`a.out` foi criado por Ken Thompson (que junto com Dennis Ritchie e Brian Kernighan criaram a linguagem C). Informações sobre este formato podem ser encontrados no site da Bell Labs².

COFF A documentação detalhada está disponível em [14]. Para informações sobre a adaptação do COFF no formato PE (Portable Executable) dos sistemas operacionais Windows veja [10].

Antes de prosseguir na descrição do formato ELF, é interessante observar o que estes formatos têm em comum. Abaixo são apresentados estes pontos em comum e como vê-los no linux utilizando programas que decodificam o formato ELF em algo mais “legível”.

- 1 eles não contêm informação própria para a leitura de humanos. Seu conteúdo não prioriza a utilização de caracteres ASCII, uma vez que são acessados unicamente por programas de computador (como montadores, ligadores e carregadores). Para confirmar isto, digite

```
> cat <nomeArquivoExecutável>
```

- 2 eles são divididos em seções sendo que cada seção contém informação específica. Algumas destas seções foram apresentados nos códigos *assembly* da primeira parte deste livro (como por exemplo, as seções `.text`, `.data` e `.bss`). Além destas, existem muitas outras. Para visualizar as seções de um arquivo executável no linux, utilize o comando

```
> objdump -s <nomeArquivoExecutável>
```

¹ acrônimo de *assembler output*

² <https://www.bell-labs.com/usr/dmr/www/pdfs/man51.pdf> acesso em 09/2015

- 3 os primeiros caracteres do arquivo são chamados de número mágico³ que indicam qual o formato (e versão) em que aquele arquivo foi codificado (e consequentemente como ele deve ser lido). Para ver o *magic number* de um arquivo executável, utilize um dos comandos abaixo:

```
> readelf <nomeArquivoExecutável> -h
> hexdump <nomeArquivoExecutável> -vC | head
```

- 4 Grande parte dos formatos incluem estruturas. Dentro destas estruturas muitas vezes existem vetores indicando locais do arquivo que tem informação importante.

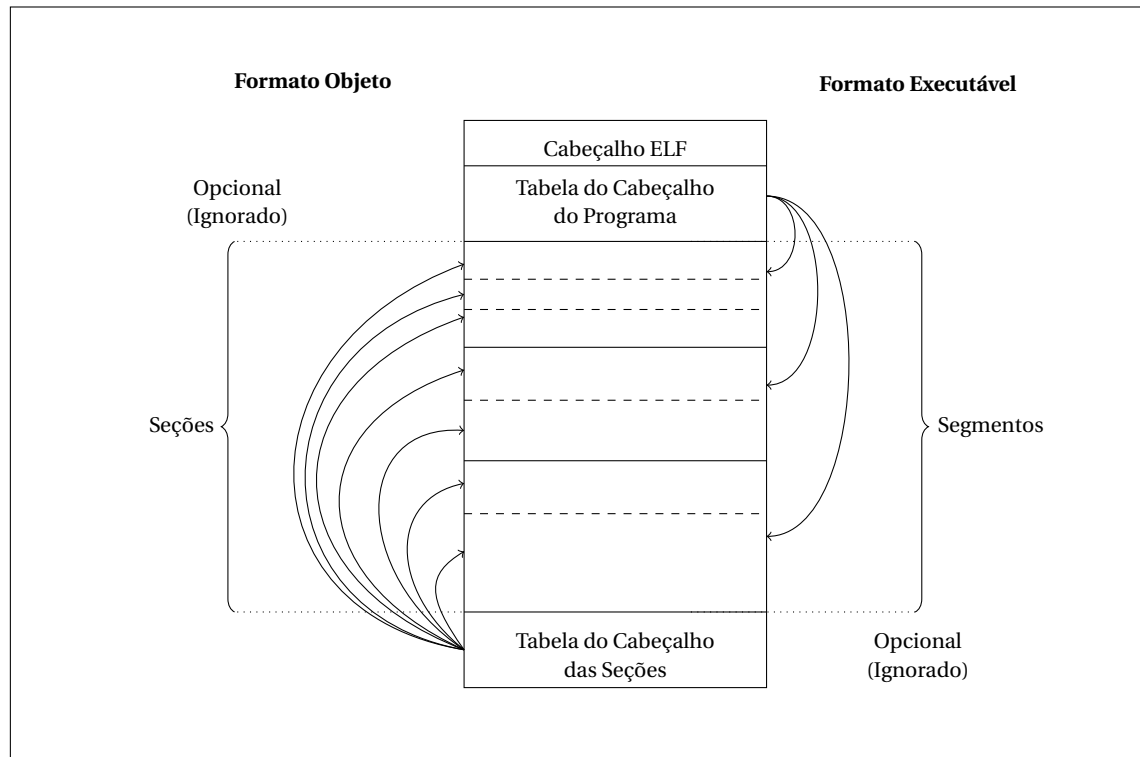


Figura 20 – Organização de um arquivo no formato ELF. Na esquerda, a organização para arquivos objeto e na direita, a organização para arquivos executáveis (adaptado de [25])

Arquivos no formato ELF representam tanto arquivos objeto como arquivos executáveis. Em ambos os casos, o início do arquivo contém um cabeçalho conforme apresentado na Figura 20. O lado esquerdo da Figura 20 mostra a organização de arquivos em formato objeto, onde:

- é obrigatória a presença da tabela de cabeçalho das seções;
- é opcional a existência da tabela de cabeçalho do programa;
- o corpo principal do arquivo é dividido em seções.

A tabela de cabeçalho das seções contém a localização (quantos bytes separam o início do arquivo, endereço zero, até o byte onde inicia a seção) de cada seção (por exemplo, `.text`, `.data`, etc) dentro do arquivo. As seções são informações pertinentes ao ligador, cujo conjunto de tarefas inclui agrupar as seções de vários arquivos objeto de entrada para gerar um arquivo executável onde as seções estão agrupadas.

O lado direito da figura mostra a organização de arquivos em formato executável, onde:

- é obrigatória a existência da tabela de cabeçalho do programa;
- é opcional a presença da tabela de cabeçalho das seções;
- o corpo principal do arquivo é dividido em segmentos.

³ *magic number*

A tabela de cabeçalho do programa contém a localização de cada segmento dentro do arquivo. Os segmentos contêm informações pertinentes para o carregador, como por exemplo, quais as permissões devem ser dadas às páginas virtuais onde aquele segmento for colocado em tempo de execução. Assim, se o segmento contiver código executável, as páginas alocadas só podem ter permissões de leitura e execução (mas não de escrita).

A Figura 20 mostra as seções separadas por traços enquanto que os segmentos são separados por linhas cheias. Considere que o arquivo representado está no formato executável. Neste caso, o segmento central é composto por duas seções, digamos `.data` e `.bss`. Esta informação pode ser usada para, por exemplo, alocá-las em páginas virtuais com as mesmas permissões. Por esta razão, o ligador as agrupa num mesmo segmento.

As estruturas dos arquivos ELF podem ser encontradas no arquivo `/usr/include/elf.h`. Por exemplo, o cabeçalho ELF64 indicado na Figura 20 é o apresentado no Algoritmo 8.1.

```

1  /* The ELF file header. This appears at the start of every ELF file. */
2
3  typedef struct
4  {
5      unsigned char e_ident[EI_NIDENT];    /* Magic number and other info */
6      Elf64_Half    e_type;                 /* Object file type */
7      Elf64_Half    e_machine;              /* Architecture */
8      Elf64_Word    e_version;              /* Object file version */
9      Elf64_Addr    e_entry;                /* Entry point virtual address */
10     Elf64_Off      e_phoff;                /* Program header table file offset */
11     Elf64_Off      e_shoff;                /* Section header table file offset */
12     Elf64_Word     e_flags;                /* Processor-specific flags */
13     Elf64_Half     e_ehsize;                /* ELF header size in bytes */
14     Elf64_Half     e_phentsize;            /* Program header table entry size */
15     Elf64_Half     e_phnum;                /* Program header table entry count */
16     Elf64_Half     e_shentsize;            /* Section header table entry size */
17     Elf64_Half     e_shnum;                /* Section header table entry count */
18     Elf64_Half     e_shstrndx;             /* Section header string table index */
19 } Elf64_Ehdr;

```

Algoritmo 8.1 – Estrutura do cabeçalho de um arquivo elf (em `/usr/include/elf.h`)

Este cabeçalho contém uma série de metadados⁴ como em qual byte do arquivo (qual *offset*) inicia o cabeçalho das seções `e_shoff` e seu tamanho (`e_shentsize`), entre outros.

8.1.1 Arquivos ELF em formato Objeto

O lado esquerdo da Figura 20 mostra a organização de arquivos em formato objeto, onde:

- é opcional a existência da tabela de cabeçalho do programa;
- o corpo principal do arquivo é dividido em seções.
- é obrigatória a presença da tabela de cabeçalho das seções;

A tabela de cabeçalho das seções contém a localização (quantos bytes separam o início do arquivo, endereço zero, até o byte onde inicia a seção) de cada seção (por exemplo, `.text`, `.data`, etc) dentro do arquivo. As seções são informações pertinentes ao ligador, cujo conjunto de tarefas inclui agrupar as seções de vários arquivos objeto de entrada para gerar um arquivo executável onde as seções estão agrupadas.

Não está mostrado na figura que cada seção contém um cabeçalho onde são armazenadas informações específicas da seção.

É importante perceber que o formato ELF não faz referência a nenhuma linguagem de programação. Isto permite a integração de programas escritos em uma linguagem, digamos C com programas escritos em outra linguagem, digamos assembly. Também permite a integração de arquivos objeto com programas em tempo de compilação. Para tal, o compilador primeiro gera o objeto do programa fonte.

Vamos agora descrever como criar, analisar e integrar arquivos ELF objeto.

Para tal, considere os programas fonte `a.c`, `b.c`, `c.c`, `d.c` e `main.c` apresentados nos Algoritmos 8.2, 8.3, 8.4, 8.5 e 8.6 respectivamente.

⁴ conjunto de dados que descrevem outros dados.

```

1 #include <stdio.h>
2 int globalA=1;
3 void a (char* s)
4 {
5     printf("%s %d\n" , s, globalA);
6 }

```

Algoritmo 8.2 – arquivo "a.c"

```

1 #include <stdio.h>
2 int globalB=2;
3 void b (char* s)
4 {
5     printf("%s %d\n" , s, globalB);
6 }

```

Algoritmo 8.3 – arquivo "b.c"

```

1 #include <stdio.h>
2 int globalC=3;
3 void c (char* s)
4 {
5     printf("%s %d\n" , s, globalC);
6 }

```

Algoritmo 8.4 – arquivo "c.c"

Não é possível gerar o executável para o arquivo main.c pois ele não tem a implementação dos procedimentos a(), b(), c() e d(). Já os arquivos a.c até d.c também não permitem geração do arquivo executável pois ele não tem a implementação do procedimento main().

```

> gcc main.c
/tmp/ccD3X8qa.o: In function 'main':
main.c:(.text+0x15): undefined reference to 'a'
main.c:(.text+0x1f): undefined reference to 'b'
main.c:(.text+0x29): undefined reference to 'c'
main.c:(.text+0x33): undefined reference to 'd'
collect2: error: ld returned 1 exit status
> gcc a.c
/usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/crt1.o: In function '_start':
(.text+0x20): undefined reference to 'main'
collect2: error: ld returned 1 exit status
>

```

Existem várias formas de combinar os arquivos para gerar o arquivo executável:

- Método 1: fonte main.c e fontes a.c, b.c, c.c e d.c.

```

> gcc main.c a.c b.c c.c d.c -o main
> ./main
dentro de a 1
dentro de b 2
dentro de c 3
dentro de d 4

```

- Método 2: fonte main.c e objeto a.o:

```

> gcc -c a.c -o a.o
> gcc -c b.c -o b.o

```

```

1 #include <stdio.h>
2 int globalD=4;
3 void d (char* s)
4 {
5     printf("%s %d\n" , s, globalD);
6 }

```

Algoritmo 8.5 – arquivo "d.c"

```

1 void a (char* s);
2 void b (char* s);
3 void c (char* s);
4 void d (char* s);
5 int main (int argc, char** argv)
6 {
7     a("dentro de a");
8     b("dentro de b");
9     c("dentro de c");
10    d("dentro de d");
11 }

```

Algoritmo 8.6 – arquivo "main.c"

```

> gcc -c c.c -o c.o
> gcc -c d.c -o d.o
> gcc main.c a.o b.o c.o d.o -o main

```

- Método 3: fontes a.c, b.c, c.c, d.c e objeto main.o:

```

> gcc -c main.c -o main.o
> gcc main.o a.c b.c c.c d.c -o main

```

- Método 4: com arquivos objeto:

```

> gcc -c main.c -o main.o
> gcc -c a.c -o a.o
> gcc -c b.c -o b.o
> gcc -c c.c -o c.o
> gcc -c d.c -o d.o
> gcc main.o a.o b.o c.o d.o -o main

```

Uma vez gerados os arquivos objeto e executável, podemos analisá-los em detalhes.

Existem vários programas capazes de listar o conteúdo dos arquivos ELF objeto. Utilizaremos o `readelf` para examinar algumas questões pontuais.

A primeira questão está relacionada com as seções. Como já explicado, as seções de um arquivo ELF objeto contêm informações úteis para o programa ligador. Isto fica mais claro no método 4 acima, onde as entradas são arquivos ELF objeto.

Conforme já explicado, a Figura 20 não indica, mas cada seção contém um cabeçalho que, por sua vez, contém informações daquela seção. Isto no arquivo `/usr/include/elf.h`:

O leitor mais paciente (MUUUUITO mais paciente) pode examinar o arquivo objeto para encontrar estas informações. Os demais leitores podem visualizar as informações contidas nos cabeçalhos das seções com o comando `readelf -S` (a opção `-S` indica que é para imprimir somente as informações que constam nos cabeçalhos de cada seção).

```
> readelf a.o --sections --wide
```

There are 13 section headers, starting at offset 0x2e8:

Section Headers:

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
------	------	------	---------	-----	------	----	-----	----	-----	----

```

1  /* Section header.  */
2
3  typedef struct
4  {
5      Elf64_Word    sh_name;           /* Section name (string tbl index) */
6      Elf64_Word    sh_type;           /* Section type */
7      Elf64_Xword   sh_flags;          /* Section flags */
8      Elf64_Addr    sh_addr;           /* Section virtual addr at execution */
9      Elf64_Off     sh_offset;         /* Section file offset */
10     Elf64_Xword    sh_size;           /* Section size in bytes */
11     Elf64_Word     sh_link;           /* Link to another section */
12     Elf64_Word     sh_info;           /* Additional section information */
13     Elf64_Xword    sh_addralign;      /* Section alignment */
14     Elf64_Xword    sh_entsize;        /* Entry size if section holds table */
15 } Elf64_Shdr;

```

Algoritmo 8.7 – Estrutura Elf64_Shdr: cabeçalho das seções (em /usr/include/elf.h)

[0]	NULL	0000000000000000	000000	000000	00	0	0	0
[1]	.text	PROGBITS	0000000000000000	000040	00002b	00	AX	0 0 1
[2]	.rela.text	RELA	0000000000000000	000220	000048	18	I 11	1 8
[3]	.data	PROGBITS	0000000000000000	00006c	000004	00	WA	0 0 4
[4]	.bss	NOBITS	0000000000000000	000070	000000	00	WA	0 0 1
[5]	.rodata	PROGBITS	0000000000000000	000070	000007	00	A	0 0 1
[6]	.comment	PROGBITS	0000000000000000	000077	000036	01	MS	0 0 1
[7]	.note.GNU-stack	PROGBITS	0000000000000000	0000ad	000000	00		0 0 1
[8]	.eh_frame	PROGBITS	0000000000000000	0000b0	000038	00	A	0 0 8
[9]	.rela.eh_frame	RELA	0000000000000000	000268	000018	18	I 11	8 8
[10]	.shstrtab	STRTAB	0000000000000000	000280	000061	00		0 0 1
[11]	.symtab	SYMTAB	0000000000000000	0000e8	000120	18		12 9 8
[12]	.strtab	STRTAB	0000000000000000	000208	000016	00		0 0 1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
 I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
 0 (extra OS processing required) o (OS specific), p (processor specific)

As opções `--sections --wide` pedem para imprimir somente os cabeçalhos das seções sem quebrar a linha.

São ao todo 13 seções. As duas primeiras colunas contêm o número e nome de cada uma das seções. A seção zero é só usada para referência. A seção 1 comporta a seção `.text`, a seção 2 comporta a seção `.rela.text` e assim por diante.

A terceira coluna (*Type*) indica o tipo da seção (o `sh_type` no cabeçalho da seção), a semântica e conteúdo daquela seção. A lista é extensa e pode ser encontrada na *manpage* do `elf`⁵. Destacamos as seguintes:

PROGBITS Indica que o conteúdo é definido pelo programa.

NOBITS A seção não ocupa espaço no arquivo.

STRTAB A seção contém uma tabela de *strings*.

SYMTAB Contém uma tabela dos símbolos disponibilizados pelo arquivo objeto.

A coluna *Flags* (`sh_flags`) contém bits cujo mapeamento é mostrado ao final da impressão. Os *flags* da seção `.text` são `AX`. O *flag* `A` (`Alloc`) significa que a seção ocupa memória durante a execução do processo enquanto que o *flag* `X` (`Execute`) diz que esta seção contém código executável.

Para o escopo deste livro, este nível de detalhamento é suficiente, pois permite explicar como o ligador trabalha. Para uma explicação mais detalhada, consulte as *man pages*, o arquivo `/usr/include/elf.h`, o livro de John Levine [25] e o manual do ELF [8].

Como será visto no Capítulo 9, uma das funções do ligador é ler de um a vários arquivos objeto e juntá-los em um único arquivo executável. Um dos cuidados que ele deve ter é garantir que exista exatamente uma implementação de cada procedimento referenciada pelo programa. Outro cuidado é juntar as seções equivalentes em vários arquivos objeto em um mesmo segmento do arquivo executável.

⁵ > `man elf`

É evidente que para efetuar estas operações, os arquivos ELF objeto devem conter as informações necessárias para tal.

8.1.2 Arquivos ELF em formato Executável

O lado direito da Figura 20 (página 108) mostra a organização de arquivos em formato executável, onde:

- é obrigatória a presença da tabela de cabeçalho do programa;
- o corpo principal do arquivo é dividido em segmentos, que podem ser agrupamentos de seções.
- é opcional a existência da tabela de cabeçalho de seções.

A tabela de cabeçalho do programa contém os endereços de início de cada segmento. Cada segmento contém informações pertinentes ao carregador, como as permissões do segmento (leitura, escrita e execução). Isto permite ao carregador alocar páginas virtuais e indicar as permissões de cada página de acordo com o indicado no segmento.

Para analisar o conteúdo de um arquivo ELF, voltaremos aos Algoritmos 8.6 E 8.2 (página 111), em especial ao seu executável main.

Assim como nas seções dos arquivos objeto, os segmentos dos arquivos executáveis também tem um cabeçalho. O cabeçalho dos segmentos está presente no arquivo `/usr/include/elf.h` e apresentado no Algoritmo 8.8.

```

1  /* Program segment header.  */
2
3  typedef struct
4  {
5      Elf64_Word    p_type;           /* Segment type */
6      Elf64_Word    p_flags;         /* Segment flags */
7      Elf64_Off     p_offset;        /* Segment file offset */
8      Elf64_Addr    p_vaddr;         /* Segment virtual address */
9      Elf64_Addr    p_paddr;         /* Segment physical address */
10     Elf64_Xword    p_filesz;        /* Segment size in file */
11     Elf64_Xword    p_memsz;         /* Segment size in memory */
12     Elf64_Xword    p_align;         /* Segment alignment */
13 } Elf64_Phdr;

```

Algoritmo 8.8 – Estrutura `Elf64_Phdr`: cabeçalho das segmentos (de `/usr/include/elf.h`)

Novamente utilizaremos o arquivo `readelf` para examinar o conteúdo de um arquivo executável, o programa main. Só iremos examinar o conteúdo dos segmentos:

```

> readelf main --segments --wide
Elf file type is EXEC (Executable file)
Entry point 0x400430
There are 9 program headers, starting at offset 64

```

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000040	0x0000000000400040	0x0000000000400040	0x0001f8	0x0001f8	R E	0x8
INTERP	0x000238	0x0000000000400238	0x0000000000400238	0x00001c	0x00001c	R	0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]							
LOAD	0x000000	0x0000000000400000	0x0000000000400000	0x0008ac	0x0008ac	R E	0x200000
LOAD	0x000e10	0x0000000000600e10	0x0000000000600e10	0x000238	0x000240	RW	0x200000
DYNAMIC	0x000e28	0x0000000000600e28	0x0000000000600e28	0x0001d0	0x0001d0	RW	0x8
NOTE	0x000254	0x0000000000400254	0x0000000000400254	0x000044	0x000044	R	0x4
GNU_EH_FRAME	0x0006e0	0x00000000004006e0	0x00000000004006e0	0x000054	0x000054	R	0x4
GNU_STACK	0x000000	0x0000000000000000	0x0000000000000000	0x000000	0x000000	RW	0x10
GNU_RELRO	0x000e10	0x0000000000600e10	0x0000000000600e10	0x0001f0	0x0001f0	R	0x1

Section to Segment mapping:

Segment Sections...

00

```

01      .interp
02      .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version \
.gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame
03      .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
04      .dynamic
05      .note.ABI-tag .note.gnu.build-id
06      .eh_frame_hdr
07
08      .init_array .fini_array .jcr .dynamic .got

```

As opções `--segments --wide` pedem para imprimir somente os cabeçalhos dos segmentos sem quebrar a linha.

A segunda linha indica qual o endereço da primeira instrução, o ponto de entrada (0x400440).

A terceira linha indica que há nove segmentos. Vamos descrever só o primeiro segmento, o segmento zero. Ele é do tipo PHDR, ou seja, é a localização da própria tabela de cabeçalhos. Sua localização dentro do arquivo (*offset*) é no byte 0x00 0040, e deve ser mapeado no endereço virtual 0x0000 0000 0040 0040. Este segmento ocupa 0x00 01f8 bytes (ou seja, dentro do arquivo, ele vai do *offset* 0x00 0040 até 0x00 0237). Na memória devem ser alocados 0x00 01f8 bytes para este segmento.

A penúltima coluna, *flags* indica que o segmento é R E, ou seja, de leitura e execução. Finalmente, a última coluna indica que ele deve ser colocado em um endereço múltiplo de oito.

O final da listagem apresenta o mapeamento de seções para segmentos. O mais interessante de descrever é o segmento 2 (LOAD). Lá estão agrupadas as seções que contêm código executável (em especial a seção `.text`) de todos os arquivos objeto. Esta organização permite ao carregador alocar uma página virtual com permissões de leitura e execução para aquele segmento (a partir do endereço virtual 0x0000 0000 0040 0000) e copiar para lá os 0x00 08ac bytes contidos no segmento 2.

8.1.3 Bibliotecas

O termo em inglês “*library*” tem um significado um pouco diferente da palavra “biblioteca”.

Em português, o termo “biblioteca” é utilizado para referenciar um conjunto de livros, manuscritos, etc.⁶ enquanto que em inglês o termo “*library*” é mais amplo⁷.

O significado do termo biblioteca usado neste livro é equivalente à definição do inglês: “coleção de coisas semelhantes”. Uma definição mais precisa é: “uma coleção de componentes de programa que podem ser reusados em vários programas” [35]. No caso deste livro, coleção contendo arquivos objeto.

A motivação da criação de bibliotecas de arquivos objeto reside na simplificação do processo de desenvolvimento de programas. Reescrever código disponível não é uma boa prática, pois além de aumentar o tempo de desenvolvimento pode incluir erros não presentes no código disponível. Por exemplo, o procedimento `printf()` existente já foi testado centenas de milhares de vezes, o que torna seu uso mais confiável do que criar uma nova versão pessoal deste procedimento.

Com o passar dos anos, muitos procedimentos foram desenvolvidos. Uma forma de disponibilizá-los aos programadores seria através de arquivos objeto como visto na Seção 8.1.1, porém a quantidade de arquivos objeto poderia ser grande, tornando inconveniente descobrir quais são os arquivos objeto que contêm a implementação de cada procedimento.

Outra solução seria agrupar arquivos objeto em coleções de arquivos chamados bibliotecas. Um problema potencial é que cada biblioteca pode conter centenas ou até milhares de arquivos objeto, tornando a busca por arquivos objeto algo demorado. Por esta razão, os arquivos biblioteca incluem também informação gerencial como índices para encontrar os arquivos objeto desejados.

⁶ Definição completa do dicionário Aurélio:

- Conjunto de livros, manuscritos, etc.
- Sala ou edifício onde está essa coleção.

⁷ definição completa do dicionário Merrian-Webster:

- a place where books, magazines, and other materials (such as videos and musical recordings) are available for people to use or borrow;
- a room in a person's house where books are kept;
- a collection of similar things (such as books or recordings);

No linux estão disponíveis três tipos de bibliotecas que se diferenciam principalmente nos seguintes aspectos:

1. na forma com que são geradas;
2. na forma com que são invocadas em tempo de execução;
3. no espaço físico que ocupam no arquivo executável e na memória.

As próximas seções descrevem cada um dos tipos de bibliotecas, destacando em especial os aspectos supracitados. A Seção 8.1.4 descreve as bibliotecas estáticas, a Seção 8.1.5 descreve as bibliotecas compartilhadas e a Seção 8.1.6 descreve as bibliotecas dinâmicas.

8.1.4 Bibliotecas Estáticas

Uma biblioteca estática é um conjunto de arquivos objeto agrupados em um único arquivo. Normalmente se utiliza a extensão “.a” para indicar uma biblioteca estática, e em Linux elas são criadas através do programa *ar* (*archive*). Este programa pode ser utilizado para concatenar qualquer conjunto de arquivos, mas o seu uso está mais frequentemente relacionado com a concatenação de arquivos objeto.

A sequência de comandos abaixo mostra como criar um arquivo biblioteca a partir de quatro arquivos objeto:

```
> ar -src libMyStaticLib.a a.o b.o c.o d.o
> gcc -o main -lMyStaticLib -L.
```

A linha de compilação para gerar um arquivo executável agora é um pouco diferente:

ar -src (-s) cria o índice de símbolos, (-r) substitui o arquivo se já existir e (-c) cria o arquivo `libMyStaticLib` se não existir.

-lMyStaticLib diz ao compilador que o código dos procedimentos que não forem encontrados nos demais arquivos podem ser encontrados na biblioteca `libMyStaticLib.a` (observe que são omitidos a extensão (.a) e os caracteres iniciais, que sempre tem que ser “lib”).

-L. Quando houver necessidade, o ligador irá procurar pela biblioteca em alguns diretórios padrão (`/usr/lib`, `/usr/local/lib`, etc), porém não irá procurar no diretório corrente. A opção `-L` avisa ao ligador para procurar pela biblioteca também no diretório indicado, que no caso é “.”, ou seja, o diretório corrente.

O programa “ar” simplesmente concatena os arquivos indicados e coloca algumas informações adicionais para compor o arquivo “`libMyStaticLib.a`”. Este mecanismo é largamente utilizado e algumas centenas de bibliotecas estáticas (extensão “.a”) podem ser encontrados a partir de `/usr/lib`⁸. Cada biblioteca contém uma série de procedimentos relacionados. Exemplos incluem `libm.a` (biblioteca de procedimentos matemáticos), `libX.a` (X window), `libjpeg.a` (compressão de imagens para formato jpeg), `libgtk.a` (interface gráfica), etc.

Cada biblioteca pode ser composta de vários arquivos objeto, e cada arquivo objeto implementa dezenas ou centenas de procedimentos, e por vezes torna-se difícil lembrar-se de todos os procedimentos e da ordem dos parâmetros em cada uma delas. Por isso, para cada arquivo biblioteca está relacionado um arquivo cabeçalho (header) e este arquivo deve ser incluído para que o compilador possa verificar se os parâmetros correspondem aos parâmetros da implementação do procedimento.

Por vezes é difícil lembrar qual o arquivo cabeçalho correto e qual a biblioteca que deve ser incluída para que um programa possa usar um determinado procedimento. Por isso, as *man pages* relacionam os nomes dos procedimentos, ao arquivo cabeçalho que deve ser incluído e à biblioteca desejada.

Tomemos como exemplo o procedimento `sqrt()`. Quando eu digito “man sqrt”, aparecem as seguintes informações:

```
SQRT(3)          Linux Programmer's Manual          SQRT(3)

NAME
    sqrt, sqrtf, sqrtl - square root function

SYNOPSIS
    #include <math.h>
```

⁸ no meu linux, a maioria deles está em `/usr/lib/gcc/x86_64-linux-gnu/`

```
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
```

Link with `-lm`.

DESCRIPTION

The `sqrt()` function returns the non-negative square root of `x`.
It fails and sets `errno` to `EDOM`, if `x` is negative.

(...)

O cabeçalho do procedimento está em `<math.h>` e também é apresentada algumas variações `sqrtf()` e `sqrtl()`. Com isso, o programador não precisa consultar `<math.h>` para saber quais e de qual o tipo são os parâmetros. Também indica como fazer a ligação (Link with `-lm`) É importante mencionar que muitas vezes não é indicado como fazer a ligação. Nestes casos, a ligação é automática, uma vez que o procedimento em questão faz parte da `libc`, biblioteca que é usada automaticamente para todos os programas escritos na linguagem “C”. Verifique isso consultando as *man pages* dos procedimentos `malloc()`, `printf()`, etc.

No programa `main.c` (Algoritmo 8.6), a verificação dos parâmetros é feita nas primeiras linhas, onde há protótipos dos procedimentos. Como não há código associado a estes procedimentos, o compilador entende que esta é a “assinatura” dos procedimentos e os usa para checar quais os parâmetros e o valor de retorno.

Para exemplificar a importância destas linhas, compile e execute o programa sem as linhas dos protótipos.

Como não há protótipo, o compilador usa a primeira instância do procedimento como assinatura. Como exemplo, considere a primeira chamada de `a()` na linha 7 do Algoritmo 8.6. Analisando esta chamada, o compilador assume que a assinatura é

```
int a(char*)
```

A diferença é que pressupõe que o procedimento retorna um inteiro. Apesar disso, o ligador irá incluir normalmente o código do procedimento e vai acreditar que a checagem de tipos foi feita.

Quando o programa for executado, várias coisas podem acontecer, inclusive o programa pode gerar o resultado correto. Porém o mais comum é ocorrer algum tipo de erro de execução. Aqui o erro é identificável porque o compilador avisa (warning) que falta o protótipo e o que ele está fazendo. Esta “permissividade” é uma das deficiências da linguagem C, apesar de muitos programadores experientes entenderem que é isso é uma virtude, pois a torna ótima para “escovar bits”.

Agora, vamos nos concentrar no formato de um arquivo *archive*. Adotaremos a terminologia de Levine [25], onde o termo “arquivo” é usado para referenciar um arquivo objeto individual e módulo é usado para referenciar um arquivo objeto dentro do arquivo biblioteca.

Um arquivo *archive* é composto por duas partes:

Cabeçalho Global Contém o *magic number* “!`<arch>`\\n”

Segmentos Cada segmento contém informações sobre um arquivo objeto. Cada segmento divide-se em duas partes:

cabeçalho A estrutura apresentada no Algoritmo 8.9, que contém informações sobre o arquivo objeto.

módulo Uma cópia do arquivo objeto ocupando `ar_size` bytes.

Com esta informação podemos iniciar a análise de um arquivo no formato *archive*. Para tal, considere os programas fonte `a.c`, `b.c`, `c.c`, `d.c` e `main.c` apresentados nos Algoritmos 8.2, 8.3, 8.4, 8.5 e 8.6 respectivamente (página 110).

O comando para gerar o arquivo *archive* a partir dos respectivos arquivos objeto é o seguinte:

```
> ar -src libMyStaticLib.a a.o b.o c.o d.o
```

O arquivo `libMyStaticLib.a` é composto pelo cabeçalho global e por quatro segmentos. Cada segmento contém informações sobre o arquivo objeto e é seguido pela cópia do arquivo objeto.

Um arquivo no formato *archive* é composto basicamente por uma concatenação de arquivos (ou módulos), e vários programas podem ser usados para percorrê-los e imprimir o conteúdo de cada módulo. Por exemplo, o comando “`readelf libMyStaticLib.a -s`” gera quase o mesmo resultado que o comando “`readelf a.o b.o c.o d.o -s`”.

```

1 struct ar_hdr
2 {
3     char ar_name[16];           /* Member file name, sometimes / terminated. */
4     char ar_date[12];          /* File date, decimal seconds since Epoch. */
5     char ar_uid[6], ar_gid[6]; /* User and group IDs, in ASCII decimal. */
6     char ar_mode[8];           /* File mode, in ASCII octal. */
7     char ar_size[10];          /* File size, in ASCII decimal. */
8     char ar_fmag[2];           /* Always contains ARFMAG. */
9 };

```

Algoritmo 8.9 – Estrutura `ar_hdr`: cabeçalho de um arquivo do tipo *archive* (de `/usr/include/ar.h`)

8.1.5 Bibliotecas Compartilhadas

Apesar de serem simples de usar e simples de criar, as bibliotecas estáticas podem causar alguns inconvenientes:

- replicação do código em disco. Os procedimentos incluídos estaticamente são copiados para dentro do arquivo executável. Um exemplo de um procedimento com estas características é o `printf()`. Quase todos os programas que a utilizam, e se todos estes a utilizarem a partir de bibliotecas estáticas, então cada um deles terá uma cópia deste procedimento amarrada em seu código executável. Considere um diretório com 10 arquivos executáveis, e que o código do procedimento `printf()` ocupa 100 bytes (só para facilitar as contas). Então, $10 * 100 = 1000$ bytes daquele diretório estarão ocupados somente pelo procedimento `printf()`.
- replicação do código em memória. Considere um mesmo procedimento está presente em vários arquivos executáveis, e que todos estes sejam colocados em execução. Isto significa que o procedimento está presente várias vezes na memória. Em linux, isto é mais facilmente visto com os procedimentos da `libc`, em especial o procedimento `printf()`.
- controle de versão. Considere um procedimento que contém um erro não detectado inicialmente. Considere também que em um ambiente linux, vários programas foram compilados com este procedimento. Quando o erro for corrigido, todos os programas que usaram a versão anterior (com erro) ainda estarão com o código errado em seus arquivos executáveis. Para incluir a versão corrigida, todos eles deverão ser recompilados.

As bibliotecas compartilhadas surgiram para minimizar estes problemas. O problema da memória é resolvido ao alocar uma única cópia dos procedimentos em memória e mapeando as páginas virtuais dos processos nas páginas reais onde os procedimentos são copiados.

Já o problema de espaço em disco é minimizado não exigindo que o código dos procedimentos seja colocado no código executável, mas sim a localização (caminho) e nome da biblioteca compartilhada.

Por fim, o problema do controle de versão é resolvido ao não incluir o código no arquivo executável e sim uma referência ao arquivo que contém o código. Ao ser executado, o programa vai até o arquivo referenciado e carrega o procedimento lá presente. Se o arquivo referenciado for corrigido, uma nova execução irá carregar este novo procedimento.

Todos os sistemas operacionais modernos permitem a criação deste tipo de biblioteca, e no linux a `libc` é incluída automaticamente como biblioteca compartilhada (e não como estática).

A sequência de comandos para gerar uma biblioteca compartilhada para os arquivos objeto `a.c`, `b.c`, `c.c` e `d.c` é a seguinte:

```

> gcc -fPIC -c -Wall a.c -o a.o
> gcc -fPIC -c -Wall b.c -o b.o
> gcc -fPIC -c -Wall c.c -o c.o
> gcc -fPIC -c -Wall d.c -o d.o
> ld -shared -o libMySharedLib.so a.o b.o c.o d.o
> gcc main.c -L. -lMySharedLib -o mainShared

```

A opção `-fPIC` gera um código objeto relocável, ou seja, um código cujos endereços são definidos na hora de carregar o programa na memória. A opção `-Wall` pede para sejam impressos todos os avisos de erro (warnings). A penúltima linha é que gera a biblioteca compartilhada com o uso da opção `-shared`. Observe que a biblioteca é gerada pelo ligador e que a extensão usada é `.so` (*shared object*). Finalmente, para gerar o programa executável, a linha é igual àquela apresentada para bibliotecas estáticas (Seção 8.1.4).

Porém, ao executar o programa, recebemos a seguinte mensagem:

```
> ./mainShared
./mainShared: error while loading shared libraries: libMySharedLib.so:
cannot open shared object file: No such file or directory
```

Esta mensagem de erro indica que o carregador não conseguiu encontrar a biblioteca compartilhada `libMySharedLib.so` (como será visto adiante, o programa executável armazena o nome do arquivo objeto mas não o diretório onde ele se encontra. Veja Exercício 8.1). Para corrigir este problema, a solução mais simples é incluir o caminho da biblioteca `libSharedMyLib.so` dentro da variável de ambiente `LD_LIBRARY_PATH`.

```
> echo $LD_LIBRARY_PATH
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
> echo $LD_LIBRARY_PATH
:.
> ./mainShared
dentro de a 1
dentro de b 2
dentro de c 3
dentro de d 4
```

A variável `LD_LIBRARY_PATH` indica o diretório do sistema onde as bibliotecas compartilhadas residem.

Para finalizar esta seção, vamos abordar brevemente o formato do arquivo objeto compartilhado. Ao contrário das bibliotecas estáticas que usam o formato *archive*, as bibliotecas compartilhadas usam o formato ELF. O motivo reside no fato de as bibliotecas compartilhadas terem de ser colocadas na memória em tempo de execução, quando as informações sobre segmentos e mapeamentos no endereço virtual são úteis.

É importante observar que existe o problema de fazer o procedimento do programa apontar para a implementação provida pela biblioteca dinâmica. Este tópico será analisado no Capítulo 10.

8.1.6 Bibliotecas Dinâmicas

Apesar de economizarem memória quando comparadas com as bibliotecas estáticas, pode ocorrer que algumas bibliotecas compartilhadas sejam incluídas no espaço de memória virtual de um processo, mas que sejam pouco utilizadas ao longo da execução. Isto também acarreta gasto desnecessário de memória.

Em ambientes onde o recurso memória é muito escasso (como por exemplo, em um computador com pouca memória), pode ser interessante exigir que sejam carregadas na memória somente as bibliotecas que serão efetivamente usadas, e que isto ocorra de acordo com as necessidades impostas pelo programador. Por exemplo, se o programa precisar de uma biblioteca no início e no fim da execução do programa, é razoável carregá-la no início, liberá-la durante a execução do programa e talvez recarregá-la ao final para terminar o programa.

As bibliotecas que se comportam desta forma são chamadas de bibliotecas dinâmicas (*Dynamic Link Libraries* - DLLs), e para usá-las é necessário um esforço maior do programador, uma vez que o programa principal deve sofrer alterações significativas. Como habitual, vamos exemplificar para uso na linguagem C em Linux.

A ideia geral é criar ferramentas que permitem que o programador possa:

1. incluir a biblioteca dinâmica em qualquer momento da execução;
2. liberar a biblioteca dinâmica em qualquer momento da execução;
3. executar os procedimentos da biblioteca dinâmica incluída.

Em linux, foi criada uma API para executar estas tarefas, que é chamada de interface para o carregador dinâmico. Os procedimentos básicos desta API são `dlopen()`, `dlsym()`, `dlclose()` e `dlerror()`.

A interface completa está documentada nas *man pages* (por exemplo, `man dlopen`), e aqui iremos apresentar um exemplo de como utilizá-las. Os passos necessários são:

1. Deve ser incluído o cabeçalho `<dlfcn.h>`, que contém os protótipos dos procedimentos `dlopen()`, `dlclose()` e `dlsym()`.
2. A biblioteca deve ser aberta explicitamente através do procedimento `dlopen`, que irá retornar um handle para esta biblioteca:

```
(...)
void *handle;
char *pathLib="/home/.../libMyDynamicLib.so";
(...)
handle = dlopen ( pathLib, RTLD_LAZY );
(...)
```

3. O procedimento desejado na biblioteca deve ser associado a um ponteiro para aquele procedimento como exemplificado a seguir:

```
(...)
void (*localSym)(parametros da funcao na bib);
(...)
localSym = dlsym (handle, "nome da funcao na bib");
```

4. O símbolo `localSym` agora é um sinônimo do procedimento indicado na biblioteca. Para executar o procedimento da biblioteca, basta usar `localSym` com os mesmos parâmetros do procedimento.

```
localSym ( parâmetros ... );
```

5. Ao terminar de usar a biblioteca, deve-se fechá-la com o procedimento `dlclose()`:

```
dlclose ( handle );
```

Suponha que temos dois arquivos fonte (a.c e b.c) com os quais deve ser criada a biblioteca `libMyDynamicLib.so`. Para criar a biblioteca e para gerar o programa executável, deve ser digitada a seguinte sequência de comandos:

```
> gcc -fPIC -o a.o -c a.c
> gcc -fPIC -o b.o -c b.c
> ld -shared -o libMyDynamic.so a.o b.o
> gcc -o main main.c -L. -lMyDynamic -ldl
```

Observe que a única diferença com relação ao processo de compilação de bibliotecas compartilhadas é a inclusão da biblioteca `-ldl`, que contém a implementação dos procedimentos da interface para o carregador dinâmico.

Este tópico está bem documentado em dois artigos disponíveis na internet, em especial em [50, 5] (que devem ser lidos).

```
1 #include <stdio.h>
2
3 void a (char* s) {
4     printf ("(a): %s\n", s);
5 }
```

Algoritmo 8.10 – Arquivo a.c

```
1 #include <stdio.h>
2
3 void b (char* s) {
4     printf ("(b): %s\n", s);
5 }
```

Algoritmo 8.11 – Arquivo b.c

Como exemplo, considere os Algoritmos 8.10, 8.11, E 8.12.

O laço contido entre as linhas 13 e 16 espera que o usuário digite a letra a ou a letra b. As linhas 18 e 19 colocam no vetor `s` o caminho completo (desde a raiz do sistema de arquivos) para o arquivo onde está a biblioteca que iremos incluir dinamicamente. Para tal é utilizado o procedimento `getenv()` (get environment variable), que coloca um ponteiro com o

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <dlfcn.h>
4
5  int main ( int argc, char** argv ){
6      char opcao;
7      void* handle;
8      void* (*funcao)(char*);
9      char* error;
10     char* nomeBib="/usr/lib/libMyDynamicLib.so";
11     char* path;
12     char s[100];
13
14     do{
15         printf("Digite (a) para uma funcao e (b) para a outra \n");
16         scanf ("%c", &opcao );
17     } while (opcao!='a' && opcao!='b');
18
19     path = getenv ("HOME");
20     sprintf(s, "%s%s", path, nomeBib);
21
22     handle = dlopen(s, RTLD_LAZY);
23     error = dlerror();
24     if ( error ){
25         printf("Erro ao abrir %s", s );
26         exit (1);
27     }
28
29     if ( opcao == 'a' )
30         funcao = dlsym(handle, "a");
31     else
32         funcao = dlsym(handle, "b");
33
34     funcao ("texto\n");
35
36     dlclose (handle);
37 }

```

Algoritmo 8.12 – Arquivo main.c

string da variável de ambiente HOME na variável local path. Mais adiante veremos que esta variável de ambiente (na verdade todas as variáveis de ambiente) está inserida no arquivo executável.

As linhas 21 até 26 abrem a biblioteca dinâmica e verificam se houve algum erro (por exemplo, biblioteca não encontrada). Se tudo correr bem, a variável error será NULL, ou seja, zero. A opção RTLD_LAZY pode ser entendida como indicador para que a biblioteca seja carregada “preguiçosamente”, ou seja, quando algum dos procedimentos contidos nela for utilizada. Observe que a biblioteca é aberta e associada a uma variável (handle). Daquele ponto em diante, todas as vezes que for utilizada a variável handle, estaremos fazendo referência ao arquivo biblioteca aberto.

As linhas 28 até 31 indicam qual dos dois procedimentos contidos na biblioteca deve ser utilizado. O comando funcao = dlsym(handle, "a"); pode ser lida da seguinte forma: Associa o procedimento a() contido na biblioteca dinâmica indicada em handle à variável funcao. Isto se faz basicamente copiando o endereço do procedimento a() para a variável funcao.

A execução está na linha 33. Observe que aqui o conteúdo da variável funcao é ou o endereço do procedimento a() ou o endereço do procedimento b(). Como os dois procedimentos têm a mesma sequência de parâmetros, não ocorrerá nenhum problema.

Por fim, a linha 35 fecha a biblioteca.

A forma de gerar o programa executável, e o resultado obtido pela execução do programa é o seguinte:

```

> gcc -fPIC -o a.o -c a.c
> gcc -fPIC -o b.o -c b.c
> ld -shared -o libMyDynamicLib.so a.o b.o
> gcc -o main main.c -L. -ldl

```



```
> ./main
Digite (a) para uma bib e (b) para a outra
a
(a): texto
> ./main
Digite (a) para uma bib e (b) para a outra
b
(b): texto
>
```

Como pode ser visto, a geração da biblioteca (`ld ...`) é igual ao que ocorre em bibliotecas compartilhadas. Porém, há algo de novo na linha `gcc -o main main.c -L. -ldl`. A opção `-ldl`, diz para incluir estaticamente a biblioteca `libdl.a`. Esta biblioteca é que contém os procedimentos `dlopen()`, `dlclose()`, `dlsym()` e `dLError()`. Ela contém muitos outros procedimentos com o prefixo `dl*`, mas o nosso exemplo só usou estes.

Para maiores detalhes de uso de bibliotecas, consulte [5, 50]. O capítulo 10 mostra como estas bibliotecas se comportam em tempo de execução.

8.1.7 Uma conversa sobre organização

À medida que cada usuário for criando mais bibliotecas, o gerenciamento da localização destas bibliotecas pode se tornar difícil. Por isso, sugere-se que cada usuário tenha um diretório `“$HOME/usr”`, com as ramificações convencionais de `“/usr”` (`“bin”`, `“lib”`, `“include”`, `“src”`, entre outras) e coloque ali aquilo que for desenvolvendo. Todas as bibliotecas deveriam ser colocadas no mesmo local, e então basta ajustar o seu `“.bashrc”` para incluir os locais apropriados das variáveis de ambiente, em especial `PATH` (diretórios onde procurar os arquivos executáveis), `C_INCLUDE_PATH` (diretórios onde procurar arquivos cabeçalho) e `LD_LIBRARY_PATH` (diretórios onde procurar por bibliotecas compartilhadas):

```
> export PATH=$HOME/usr/bin/:$PATH
> export C_INCLUDE_PATH=$HOME/usr/include/:$C_INCLUDE_PATH
> export LD_LIBRARY_PATH=$HOME/usr/lib/:$LD_LIBRARY_PATH
```

Observe que as linhas acima fazem com que a busca inicie pelos diretórios locais (ou seja, pelo que foi desenvolvido pelo usuário) para depois procurar nos locais convencionais do sistema. Assim, se o usuário desenvolveu um programa que tem o mesmo nome de um programa do sistema, por exemplo, `cat`, o arquivo selecionado será o local (`$HOME/usr/bin/cat`) e depois o do sistema (`/usr/bin/cat`).

Exercícios

- 8.1 Os arquivos executáveis também armazenam o conjunto de arquivos objeto compartilhável necessários para a execução. Esta informação está na seção da tabela de símbolos. Utilize o programa `readelf` para visualizar todos os símbolos no arquivo executável.
- 8.2 Faça o mesmo nos arquivos objeto. Identifique os símbolos exportados pelos arquivos objeto e os símbolos importados (inclua o `main.o`).
- 8.3 Altere o Programa 8.12 para:
 - a) permitir que o usuário digite o procedimento e que responda se o procedimento está presente ou não na biblioteca;
 - b) incluir bibliotecas diferentes;
- 8.4 o comando `readelf -s` lista os símbolos presentes em um arquivo objeto. Na `libc`, encontrei 320 acertos para o procedimento `“printf()”` e só 5 para `“_printf()”`. Explique por que.

9. O Carregador e o Ligador Estático

Este capítulo trata de um tipo de carregador e dos respectivos ligadores que este texto classifica como “simples”, e apresenta os fundamentos que serão utilizados no Capítulo 10.

Os carregadores abordados aqui são aqueles que são capazes de colocar em execução um arquivo executável diretamente, sem ter de carregar arquivos objeto externos (como objetos compartilhados ou dinâmicos), ou seja, aqueles que contêm todas as informações de todos os símbolos utilizados durante a execução no próprio arquivo executável.

O termo “símbolo” será muito utilizado aqui. Corresponde ao nome dos procedimentos, variáveis locais e globais do programa fonte e aos respectivos endereços nos arquivos objeto e executável.

Este capítulo está organizado da seguinte forma: inicialmente será apresentado o funcionamento de carregadores (Seção 9.1). Esta seção apresenta vários modelos, finalizando com arquivos executáveis ELF, utilizado na primeira parte deste livro que descreveu sua organização na memória, mas que não como o carregador coloca as informações lá.

Para trabalhar, o carregador primeiramente lê um arquivo em formato executável. Este formato é bastante rígido e normalmente não é criado manualmente. A ferramenta que o cria é chamada ligador, que é o assunto da Seção 9.2. Nesta seção será descrito o funcionamento dos ligadores que geram os arquivos executáveis com informações de todos os símbolos necessários para a execução.

9.1 Carregadores Estáticos

A função do programa carregador é trazer um programa para a memória e colocá-lo em execução. O carregador é um programa associado ao sistema operacional, e cada sistema operacional trabalha normalmente com um só formato de arquivo executável específico para aquele sistema operacional. Apesar disto alguns sistemas operacionais podem trabalhar com mais de um formato (como no caso do Windows para os formatos COM, EXE e PE).

Explicado de maneira simplista, o programa carregador recebe o nome de um arquivo a ser executado, e:

1. lê o arquivo;
2. verifica se está no formato executável, e se não estiver, dá uma mensagem de erro e não prossegue;
3. cria um novo processo;
4. aloca memória para este processo;
5. copia o arquivo para o espaço de memória do processo;
6. disponibiliza o processo para a execução.

Algumas observações importantes:

- O programa não obrigatoriamente iniciará a execução assim que for colocado em execução. Em linux, por exemplo, ele é colocado na lista de processos a ser executados e o sistema operacional o selecionará para a execução eventualmente, de acordo com alguma política do escalonador de processos.
- Em princípio, um arquivo executável que é gerado para operar em uma determinada arquitetura (CPU+SO), não pode ser executado em outra arquitetura ou em outro SO. Porém existem formas de fazer isso através de emuladores, interpretadores ou ainda adaptadores. Estas ferramentas serão analisadas no Capítulo 11.
- É possível gerar código executável para uma arquitetura CPU diferente daquela na qual se está trabalhando.

Carregadores estáticos é o nome dado aos carregadores que colocam um arquivo executável que não depende de nenhum outro arquivo para a execução. Neste tipo de arquivo executável, todos os procedimentos e variáveis utilizadas pelo programa estão incluídos no arquivo executável. Os arquivos executáveis em questão são aqueles que utilizam somente bibliotecas estáticas apresentadas na Seção 8.1.4.

Desde o surgimento de computadores vários tipos de carregadores foram propostos. Como é natural, os primeiros carregadores eram muito mais simples e os carregadores modernos, principalmente por que não tinham que lidar com múltiplos processos em execução simultânea e nem com múltiplos usuários.

Este livro apresenta três tipos de carregador estático, que de certa forma acompanham a evolução dos carregadores ao longo do tempo. O primeiro tipo são os carregadores mais simples, que copiam os programas em memória física sem

relocação (Seção 9.1.1). O segundo tipo são os carregadores que copiam os programas em memória física com relocação (Seção 9.1.2). O terceiro e último tipo apresentado aqui são os carregadores que copiam os programas em memória virtual (Seção 9.1.3), como o ELF.

9.1.1 Carregadores que copiam os programas em memória física sem relocação

Esta classe de carregador corresponde ao modelo mais simples, e foi vastamente utilizada nos primeiros computadores e também nos primeiros computadores pessoais. Um dos membros mais conhecidos desta classe é o carregador que trabalha com o formato `.COM`¹, originário do sistema operacional CP/M² e adaptado para o MS-DOS. Esta seção descreve o formato `.COM` como modelo dos carregadores que copiam os programas em memória física sem relocação.

Arquivos executáveis `.COM` não tem metadados, ou seja, o seu conteúdo é composto unicamente por uma sequência de bytes que correspondem à imagem do programa em execução.

A ação do carregador do formato `.COM` é simplesmente copiá-lo para a memória e escrever no contador de programa (PC) o endereço da memória onde a execução deve começar.

Vamos nos aprofundar no funcionamento dos carregadores que trabalham com o formato `.COM`, mas antes é necessário contextualizar este modelo com a tecnologia da época.

O primeiro aspecto a destacar é que o modelo `.COM` foi projetado para trabalhar com máquinas de 16 bits, onde os programas eram limitados a 64K (2^{16}) bytes. Na realidade, o espaço era ainda menor, pois algumas regiões eram reservadas para propósitos específicos como apresentado na Figura 21.

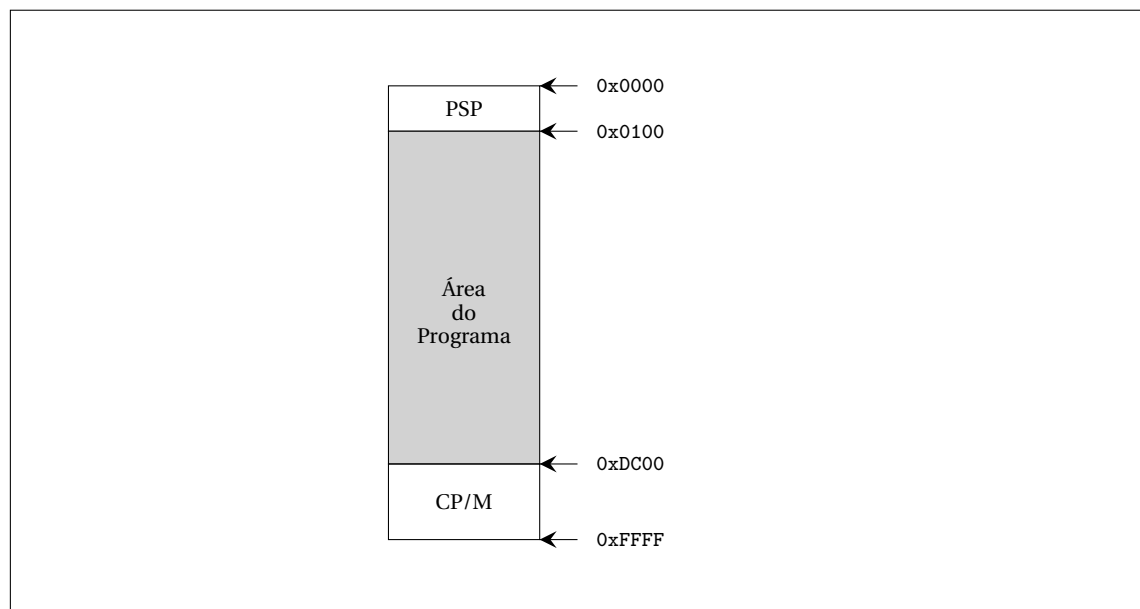


Figura 21 – Mapa de memória com um processo no formato COM em execução

A figura mostra que a memória física era dividida em três partes:

[0x0000:0x0100) A região PSP³ é uma estrutura de dados com 256 bytes que armazena informações sobre o estado de execução de um programa, como por exemplo, a linha de comando digitada, o endereço de retorno (para dentro da *shell* do CP/M) entre outros.

[0x0100:0xDC00) Região de 56.064 bytes para onde o programa executável era copiado;

[0xDC00:0xFFFF) Região de 9.216 bytes onde era armazenado o programa *shell* do CP/M.

Ao ser invocado, o carregador do CP/M executava as seguintes ações:

- Buscava o arquivo contendo o programa a ser colocado em execução;

¹ Alguns dizem que a sigla COM significa *Copy On Memory*

² *Control Program for Microcomputers*

³ *Program Segment Prefix*

- Preenchia a estrutura de dados PSP com as informações recebidas (por exemplo, a linha de comando com parâmetros);
- Copiava o arquivo a partir do endereço 0x0100 da memória, ou seja, o endereço 0x0000 do arquivo era copiado para o endereço 0x0100 da memória, o endereço 0x0001 para o endereço 0x0101 da memória e assim por diante.
- Após copiar todo o arquivo para a memória, colocava o endereço 0x0100 no registrador PC (*Instruction Pointer* no 8086) de onde o programa iniciava a execução.

Como pode ser visto, este modelo de execução era bastante simples, mas criava algumas restrições importantes:

1. o programa executável não podia ser maior do que 56.320 bytes;
2. dois programas não podiam ser colocados em execução simultânea. Como todo programa devia ser copiado a partir do endereço 0x0100 da memória, um segundo programa só podia ser colocado em execução se fosse copiado sobre o primeiro.

O segundo problema poderia ser contornado se fosse possível colocar outro programa em outra região de memória, e executá-lo lá. Porém, os programas no formato .COM partiam de algumas premissas que impediam que eles fossem executados em outros lugares. Por exemplo, um programa só podia ser executado no espaço entre os endereços 0x0100 e 0xDCFF. Esta condição fazia com que os arquivos executáveis fossem gerados com endereços fixados, aqueles que serão utilizados em tempo de execução.

Para exemplificar isto, considere a Figura 22. O lado esquerdo da figura corresponde a um arquivo em formato executável no formato .COM. Este arquivo tem 0x0200 bytes com o endereço zero do arquivo indicado acima e o endereço 0x0200 abaixo. Esta distância entre o início do arquivo até o elemento é chamado *offset*⁴. Formalmente, o termo *offset* indica a distância entre o início de um objeto (no caso, o arquivo) e o elemento de interesse. Porém também pode ser usado para indicar a distância entre o início de um objeto dentro do arquivo e o elemento de interesse.

Observe que está destacado um desvio (`jmp 0x0270`) dentro do código. É curioso que o destino do desvio seja um *offset* que está **FORA** do arquivo executável.

Para entender o porquê disto, é necessário lembrar como o carregador .COM trabalha: ele copia o arquivo a partir do endereço 0x0100 da memória física, e no caso de exemplo até o endereço 0x0300 da memória, criando a imagem de execução na memória (lado direito da Figura 22). O termo “imagem” é apropriado, pois antes de iniciar a execução, o espaço de memória entre os endereços 0x0100 e 0x0300 é uma imagem exata do arquivo (inclusive o valor das variáveis globais).

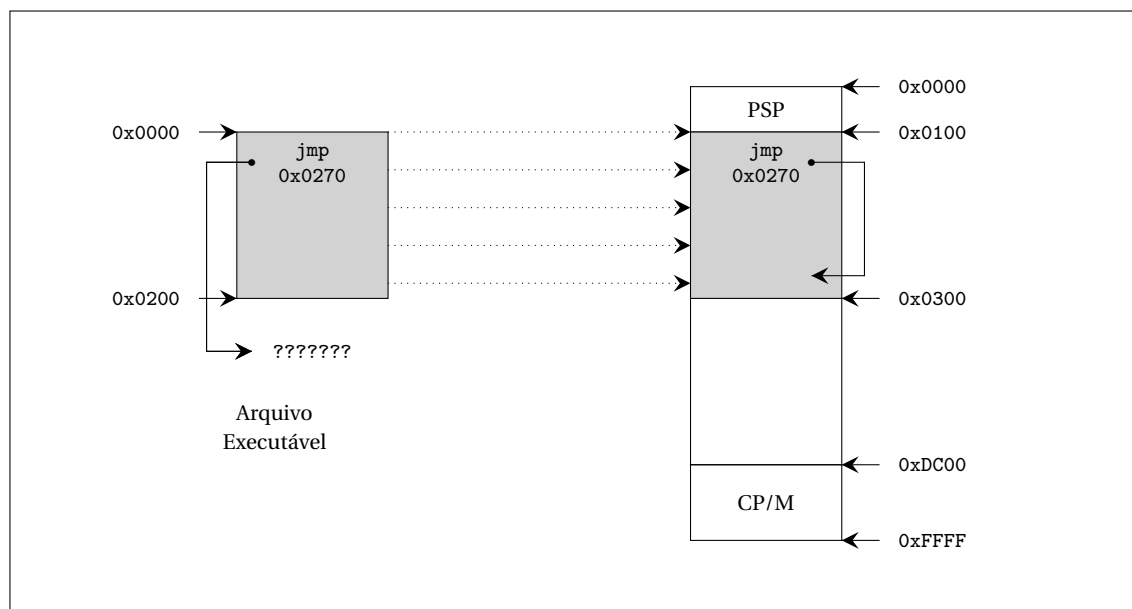


Figura 22 – Mapeamento de um arquivo para execução em memória física

⁴ optamos pela palavra em inglês. Os sinônimos em português, “contrabalançar”, “compensar” e “indenizar” não são apropriados.

Como o arquivo executável foi criado já sabendo que o endereço 0x0000 do arquivo seria mapeado no endereço 0x0100 da memória, os parâmetros das instruções assembly que acessam a memória foram todos incrementados de 0x0100. Por esta razão, na imagem de execução, a instrução `jmp 0x0270` acessa um endereço contido dentro do espaço de execução do programa na memória.

Porém, considere agora que este programa seja carregado a partir de outro endereço, digamos 0x1000. Então, quando o programa chegar no comando `jmp`, o fluxo será desviado para 0x0270, e vai seguir a execução a partir daquele ponto (sabe-se lá o que ele encontrar...).

Após este exemplo já é possível examinar algumas características deste modelo:

1. O CP/M era um sistema operacional monousuário e monoprocesso. Isto significa que não mais do que um processo podia estar em execução em cada momento. Outro processo só podia entrar em execução se o processo atual fosse cancelado (ou copiado temporariamente para outro local).
2. O carregador `.COM` não alterava nada do arquivo executável ao colocá-lo em execução, fazendo um trabalho bem simples de cópia. Ele não precisava fazer mais nada, pois já sabia que este era o único processo a ser executado e que ele começa sempre no endereço 0x0100, e que por isto os endereços de todos os símbolos (incluindo rótulos de desvio) já foram todos calculados quando da criação do arquivo executável (ou seja, em tempo de ligação).
3. Em versões posteriores do CP/M, tentou-se criar maneiras de colocar mais de um processo em execução. Um dos mecanismos consistia em chamar o sistema operacional quando determinadas teclas fossem acionadas pelo usuário (por exemplo, <ALT><TAB>). O sistema operacional por sua vez mostrava uma lista de processos suspensos que o usuário podia ativar. Quando um processo era selecionado pelo usuário, toda a imagem do processo em execução era copiada para uma área de salvamento em disco (ou de memória), para que outro processo fosse colocado em execução em seu lugar. Quando o usuário quisesse colocar o primeiro processo de volta para a execução, bastava usar as duas teclas mágicas para que a imagem do processo atual fosse salvo em disco e que a imagem salva do primeiro processo fosse copiada novamente para região de execução de memória. Com isso, criava-se a ilusão de um ambiente multiprocessos onde o mecanismo de troca de processos era gerenciado pelo usuário.
4. Para que um processo pudesse ser colocado em execução, deveria haver memória física suficiente para comportá-lo. O MS-DOS trabalhava com um dos primeiros processadores intel x86, onde se podia endereçar até 64Kbytes. Sendo assim, nenhum programa executável podia ser maior do que 64K. Posteriormente esta limitação foi contornada, porém algumas características importantes (como tamanho do segmento de 64k) continuaram.

O formato `.COM` é o mais simples de todos os formatos apresentados neste livro. Alguns aspectos dele foram projetados para trabalhar com processadores de 16 bits. Os avanços tecnológicos exigiram melhorias no modelo de execução como será visto a seguir.

9.1.2 Carregadores que copiam os programas em memória física com relocação

As deficiências no modelo `.COM` eram graves, e algumas tinham raízes no fato do processador de 16 bits acessar até $2^{16} = 64\text{Kbytes}$.

O processador intel 8086 tinha um barramento de endereços de 20 bits, que permitia endereçar até $2^{20} = 1\text{Mbytes}$, mas seus registradores tinham somente 16 bits. Como um registrador pode endereçar somente $2^{16} = 64\text{Kbytes}$, foi encontrado um mecanismo “criativo”⁵ para combinar dois registradores para endereçar o espaço de 1Mbytes.

Este mecanismo criativo permitia que o acesso à memória fosse dividido em duas partes: a primeira corresponde a um endereço fixo de memória (a base do segmento) e a segunda parte indica o deslocamento dentro daquele segmento. A notação refletia esta divisão. Por exemplo, `DS:AX` diz que o endereço real é obtido ao somar o conteúdo do registrador de segmento `DS` com o conteúdo do registrador `AX`. O termo “criativo” se justifica porque esta estranha soma de dois registradores de 16 bits resulta num endereço de 20 bits. Para mais informações consulte [34].

Com a popularização dos computadores pessoais utilizando o sistema operacional CP/M (ou MS-DOS, que usava uma abordagem semelhante), a limitação de um único processo em execução teve de ser contornada, assim como a restrição de tamanho dos programas.

Já que os computadores podiam endereçar até 1Mbytes, o espaço de memória entre 0x00100 e 0xFFFF estava disponível, e a questão era como fazer com que programas o utilizassem. Como o modelo `.COM` era muito rígido, foi criado um novo modelo de execução que foi popularizado no sistema operacional MS-DOS versão 2.0 da Microsoft.

⁵ eu optei por não utilizar o termo “gambiarra” nem o termo mais preciso “tremenda gambiarra” para não ofender os meus amigos engenheiros.

Este novo formato será aqui denominado .EXE em função da extensão dos arquivos executáveis que adotavam este modelo, mas ele também é conhecido como formato “MZ” em função do número mágico contido no cabeçalho⁶.

O termo “número mágico” corresponde a uma constante armazenada no início do arquivo para identificar a que classe ele pertence. Por exemplo, um arquivo iniciado com os caracteres #! indica que o arquivo é um *script* Shell. O Linux utiliza números mágicos para identificar o arquivo ao invés de utilizar a extensão do nome do arquivo (".odt", ".o", ".sh", etc.). Porém, é uma boa ideia utilizar extensões para que os usuários também saibam que tipo de arquivo é aquele sem terem de recorrer ao programa file.

Enquanto que no formato .COM o arquivo executável era a imagem do processo no início da execução, o formato .EXE apresentava metadados (dados sobre outros dados), que no caso era um cabeçalho com algumas informações que orientavam o carregador a colocá-lo em execução.

O Algoritmo 9.1 apresenta os metadados contidos no cabeçalho de um arquivo executável no formato .EXE no formato de uma estrutura (struct) em C.

```

1 struct EXE
2 {
3     char signature[2] = "MZ"; // {magic number}
4     short lastsize;           //{bytes used in last block}
5     short nblocks;            //{number of 512 byte blocks}
6     short nreloc;             //{number of relocation entries}
7     short hdrsize;            //{size of file header in 16 byte paragraphs}
8     short minalloc;           //{minimum extra memory to allocate}
9     short maxalloc;           //{maximum extra memory to allocate}
10    void far *sp;              //{initial stack pointer}
11    short checksum;           //{ones complement of file sum}
12    void far *ip;              //{initial instruction pointer}
13    short relocpos;            //{location of relocation fixup table}
14    short noverlay;            //{0verlay number, 0 for program}
15    char extra[];              //{extra material for overlays, etc.}
16    void far *relocs[];        //{relocation entries, starts at relocpos}
17 };

```

Algoritmo 9.1 – Formato de um cabeçalho do formato .EXE[25]

Destes campos, destacamos:

número mágico (linha 3) o número mágico ("MZ"), informa ao ligador que o arquivo executável está no formato .EXE, e que os procedimentos de carregamento de um arquivo deste formato devem ser adotados;

alocação de memória inicial As linhas 8 e 9 contêm informação sobre quanto espaço de memória deve ser alocado para este programa. Se a memória disponível fosse menor do que o mínimo exigido pelo programa, o carregador dava uma mensagem avisando do fato e não carrega o programa;

valores iniciais de registradores As linhas 10 e 12 indicam os valores iniciais de dois registradores. O fato de incluir o valor inicial do IP (*instruction pointer*) mostra uma grande diferença quanto ao formato .COM, pois o programa não precisa ser iniciado no endereço 0x0100.

código O código do programa começa imediatamente após a última instância da tabela de relocação *relocs[].

A informação mais interessante é a tabela de relocação. Esta tabela indica os locais do arquivo que têm endereços a serem modificados quando o programa for colocado em execução.

No dicionário, o termo relocação significa modificação das constantes de endereço, com a finalidade de compensar uma mudança na origem de um módulo, de um programa ou de uma seção de controle [29].

Quando esta definição é aplicada aqui, significa ajustar os endereços indicados em um arquivo executável para os endereços efetivos em memória física. Como exemplo, considere a Figura 23. Ela é muito parecida com a Figura 22, porém na instrução `jmp 0x0170`, o endereço 0x0170 não corresponde a um endereço físico de memória, mas sim a um endereço físico no código do programa contido no arquivo. A parte em cinza indica o código do programa e o endereço 0x0170 é um *offset* dentro desta área cinza.

Observe que a figura apresenta dois endereços zero. O primeiro zero ((0x0000)_A) corresponde ao endereço do primeiro byte do arquivo enquanto que o segundo zero ((0x0000)_E) corresponde ao endereço do primeiro byte do código executável, aquilo que será copiado na memória. Neste exemplo, o cabeçalho ocupa 0x0040 bytes e o código executável

⁶ Curiosamente, “MZ” são as iniciais de Mark Zbikowski, um dos projetistas do MS-DOS.

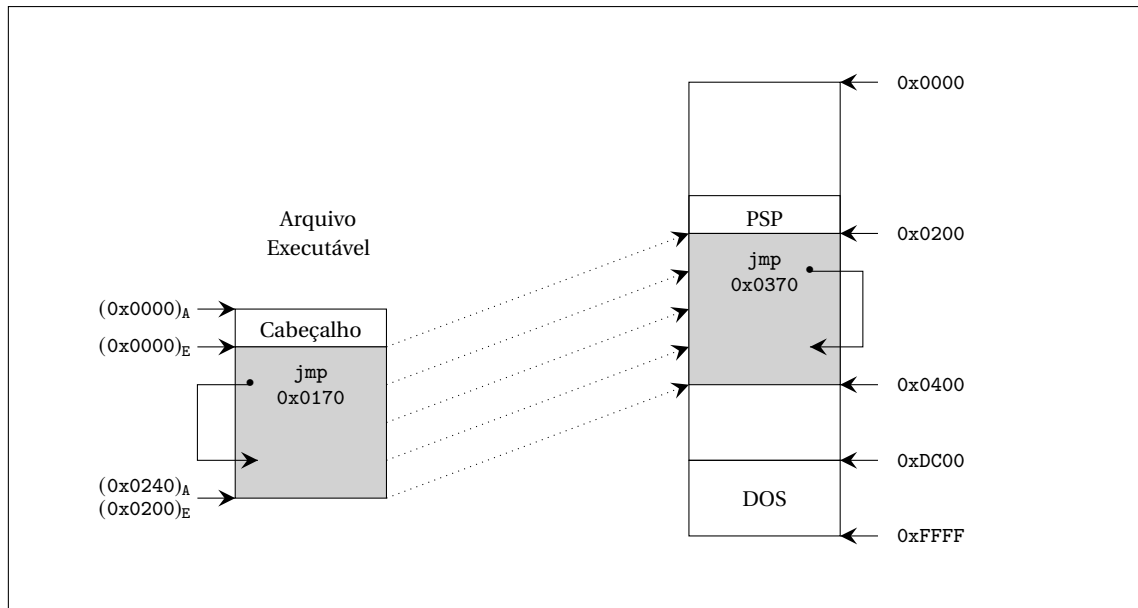


Figura 23 – Relocação de um arquivo no formato .EXE

ocupa 0x0200 bytes. Por esta razão, o último endereço do código executável é (0x0200)_E que está armazenado no último endereço do arquivo, (0x0240)_A. Observe que o código está destacado na figura e vai do endereço (0x0000)_E até (0x0200)_E, ou (0x0040)_A até (0x0240)_A.

O *offset* dentro do arquivo considera o tamanho do cabeçalho. Como o cabeçalho do Algoritmo 9.1 ocupa 0x40 bytes, o início do programa executável está no *offset* 0x0040 do arquivo.

Já o *offset* dentro do código executável não considera o tamanho do cabeçalho, ou seja, considera unicamente o deslocamento dentro do código executável. Assim, o *offset* 0x0010 do código executável está localizado no *offset* 0x0050 do arquivo executável.

No modelo .EXE, o cabeçalho contém uma lista de *offsets* dentro do código executável a serem relocados denominados *fixups*. Após o carregador copiar o arquivo na memória a partir de um endereço físico, digamos 0x1234, todos os *fixups* devem ser somados de 0x1234 para que o programa funcione no novo local.

A informação de quantos endereços relocar e os respectivos *offsets* está descrita no cabeçalho do formato .EXE. A quantidade de *offsets* está na linha 6 e a lista de *offsets* na linha 16 do Algoritmo 9.1.

Após explicar alguns aspectos do cabeçalho de um arquivo no formato .EXE, podemos apresentar o que o carregador de um programa executável neste formato faz[25]:

1. Lê o cabeçalho e verifica o número mágico para validação;
2. Encontra um local de memória apropriado para copiar o código do programa;
3. Cria a região PSP;
4. Copia o código do programa imediatamente após o PSP. Os campos *nblocks* e *lastsize* indicam o tamanho do código.
5. Executa a relocação adicionando a base do código a cada *fixup* da tabela.
6. Atribui o novo valor da pilha (*sp*) e o novo valor de *ip* (relocando-os) e inicia a execução do programa.

Uma característica importante desta classe de carregadores é que quase toda a área de memória era considerada válida para execução. O usuário podia acessar (por querer ou sem querer) locais onde hoje se prega que não devem ter acesso, como por exemplo, ao sistema operacional ou vetor de interrupção, podendo alterá-los. Para confirmar, basta ver a quantidade de livros que foram lançados na época explicando como alterar os destinos das interrupções contidas no vetor de interrupções.

Este foi uma das questões que permitiu o desenvolvimento de diversos programas mal-intencionados (e por vezes divertidos), chamados vírus. Quando o uso deste modelo foi estendido para redes de computadores, foi possível escrever vírus que se reproduziam em todos os computadores da rede (o que raramente era divertido).

Porém, é importante destacar que este modelo foi projetado para operar em um único computador que só era usado por um único usuário. Neste ambiente, é natural acreditar que o usuário é bem intencionado (afinal, porque atacar o próprio computador?). Porém, quando este sistema foi estendido para multiusuários e para redes de computadores, a inexistência de um modelo de segurança cobrou seu preço.

9.1.3 Carregadores que copiam os programas em memória virtual

Como foi visto na seção anterior, o mapeamento de um programa diretamente em memória física pode ocasionar problemas, dentre os quais se destaca o acesso a toda a memória física através de qualquer programa.

O modelo de memória virtual soluciona este problema, uma vez que cada programa recebe uma grande quantidade de memória virtual linear para trabalhar (uma grande caixa de areia onde ele pode fazer o que quiser). Como cada processo só tem acesso às suas páginas virtuais, ele fica impedido de acessar as páginas de outro.

Esta seção utiliza fortemente o conceito de memória virtual com paginação. Sugerimos a leitura do Apêndice D, em especial a Seção D.2 antes de prosseguir.

Observe que cada processo tem um grande espaço virtual de endereçamento, e que só ele pode acessar este espaço. De certa forma, isto lembra o modelo de execução .COM, onde todo o espaço de memória está disponível para ele. A diferença é que no modelo .COM a memória disponível é a memória física (que deve ser compartilhada por todos os processos) enquanto que aqui a memória disponível é virtual, uma para cada processo.

Os formatos de execução que foram criados para uso em memória virtual utilizam metadados que sugerem ao carregador como utilizar devidamente a memória virtual.

No caso do linux, os arquivos executáveis estão no formato ELF que foi descrito na Seção 8.1.2. Como o linux adota a paginação como mecanismo de implementação de memória virtual, as informações incluídas no arquivo executável incluem informações importantes para o mapeamento dos segmentos do arquivo executável em páginas virtuais.

A Figura 24 reproduz a Figura 20 da página 108.

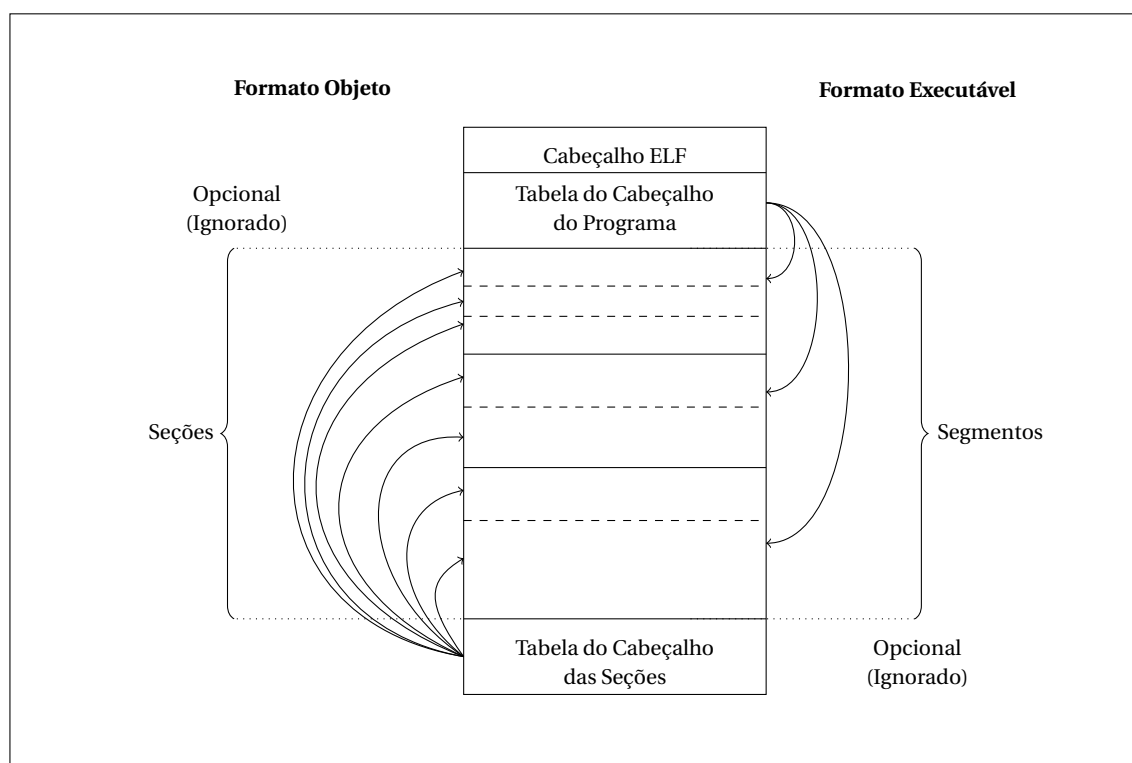


Figura 24 – Organização de um arquivo no formato ELF. Na esquerda, a organização para arquivos objeto e na direita, a organização para arquivos executáveis (adaptado de [25])

O formato ELF executável está organizado em segmentos de tamanho variável (lado direito da figura). A memória virtual está organizada em páginas de tamanho fixo. Assim, uma das tarefas do carregador ELF é mapear os segmentos

contidos em um arquivo ELF executável em páginas virtuais.

Como separar um arquivo executável em segmentos é alvo de discussão na Seção 9.2. O presente capítulo considera que o arquivo executável já existe e que já está dividido em segmentos.

A organização do arquivo ELF executável foi discutida na Seção 8.1.2. Os segmentos são compostos por um cabeçalho (metadados) e pelo conteúdo binário que será copiado para as páginas virtuais.

Cada cabeçalho de segmento contém as informações abaixo (reproduzido da página 109).

```

1  /* Program segment header. */
2
3  typedef struct
4  {
5      Elf64_Word    p_type;           /* Segment type */
6      Elf64_Word    p_flags;         /* Segment flags */
7      Elf64_Off     p_offset;        /* Segment file offset */
8      Elf64_Addr    p_vaddr;         /* Segment virtual address */
9      Elf64_Addr    p_paddr;         /* Segment physical address */
10     Elf64_Xword    p_filesz;        /* Segment size in file */
11     Elf64_Xword    p_memsz;         /* Segment size in memory */
12     Elf64_Xword    p_align;         /* Segment alignment */
13 } Elf64_Phdr;
```

Algoritmo 9.2 – Estrutura Elf64_Phdr: cabeçalho das segmentos (de /usr/include/elf.h)

Quando o carregador encontra um segmento, ele analisa o cabeçalho para saber quais atributos devem ter a página virtual que irá armazenar os dados binários daquele segmento. Como exemplo, considere que o carregador encontra um segmento com instruções (por exemplo, seção `.text`). Conforme visto na Seção 2, em tempo de execução, a seção `.text` inicia no endereço virtual `0x0000000004000000`, e este será o valor do campo `p_vaddr` (veja impressão dos segmentos de um arquivo executável com o comando `readelf` na página 113).

Os arquivos ELF executáveis podem ser carregados de forma diferente dependendo da presença de objetos compartilhados ou dinâmicos (normalmente contidas em bibliotecas compartilhadas ou dinâmicas), mas como esta seção lida somente com o carregador ELF para objetos estáticos, vamos ignorá-los por enquanto.

Os carregadores ELF abordados aqui tratam arquivos executáveis ELF que não dependem de nenhum objeto externo. Todas as funções e variáveis estão implementados dentro do próprio arquivo, como explicado na Seção 8.1.4 para bibliotecas estáticas.

O algoritmo do carregador para objetos estáticos pode ser simplificada explicada da seguinte forma:

1. leia o cabeçalho ELF (Algoritmo 8.1, na página 109);
2. aloque espaço de memória virtual linear para o processo.
3. para cada segmento do arquivo executável, faça:
 - a) leia o cabeçalho daquele segmento;
 - b) selecione páginas virtuais para o segmento começando no endereço virtual `p_vaddr` com `p_memsz` bytes;
 - c) atribua permissões às páginas virtuais selecionadas usando `p_flags`. Por exemplo, no caso do segmento contendo a seção `.text`, as permissões indicadas no segmento são de leitura e execução (mas não de escrita);
 - d) copia os dados binários do segmento (que começam no endereço `p_offset` do arquivo e que tem `p_filesz` bytes) para as páginas virtuais selecionadas;
4. ajuste informações de tempo de execução como, por exemplo, argumentos e variáveis de ambiente;
5. coloque o `e_entry` (cabeçalho ELF) em `%rip`;
6. coloque o processo na fila de execução.

9.2 Ligador para objetos estáticos

Como apresentado na Figura 19 (página 101), o programa ligador recebe como entrada um ou mais arquivos em formato objeto e gera como saída um único arquivo executável.

As tarefas executadas pelo ligador tem uma relação muito próxima com o carregador, uma vez que ele pode (e deve) preparar o arquivo executável para tornar mais rápido o carregamento do programa.

Vamos começar a discussão seguindo a mesma divisão lógica apresentada na Seção 9.1. A Seção 9.2.1 descreve o ligador para o modelo .COM, onde as tarefas do ligador são mínimas, e por vezes inexistentes. A Seção 9.2.2 descreve o ligador para o modelo .EXE, onde é necessário fazer a relocação de endereços. Por fim, a Seção 9.2.3 apresenta um ligador mais complexo para o modelo ELF.

9.2.1 Ligadores Simples

Na categoria de “ligadores simples” incluímos aqueles ligadores que fazem pouca coisa, normalmente porque não há a definição do formato de arquivo objeto. Como atualmente são poucos os ambientes onde este modelo é utilizado, vamos descrever o ambiente de trabalho utilizado pelo sistema operacional MS-DOS. Infelizmente, pouco material resistiu ao tempo, e a maior parte do relato abaixo foi baseado em minha memória. Se o leitor encontrar material que confirme (ou refute) o que escrevi, peço que me avise para que eu possa fazer as devidas correções.

Naquela época era frequente desenvolver programas utilizando ambientes integrados de desenvolvimento⁷. Nestes ambientes era possível editar os programas, compilar para gerar o arquivo executável ou ainda compilar para execução imediata⁸.

Um dos ambientes que eu usei chamava-se Turbo Pascal desenvolvido pela empresa Borland. Eu lembro que não era possível utilizar compilação separada, mas o ambiente permitia incluir arquivos fonte.

Não me lembro da existência de arquivos objeto, só dos executáveis. As pesquisas que fiz indicam que o ambiente turbo pascal gerava código executável unicamente via programa fonte.

Isto ocorria porque CP/M e MS-DOS definiram o formato dos arquivos executáveis, mas não o formato dos arquivos objeto. Sendo assim, o compilador que eu usei na época não gerava código objeto. Na época eu não dava muita importância à compilação separada nos PCs - até porque os programas eram muito pequenos e não tinham grande apelo comercial.

Como não havia o código objeto intermediário, o compilador fazia também o papel de ligador e criava o arquivo executável que seria copiado em memória física a partir do endereço base 0x0100, e os demais endereços eram calculados a partir dali.

9.2.2 Ligadores Com Relocação

O modelo citado na seção anterior não podia ser aplicado para gerar executáveis no formato .EXE. Aqui já é possível distinguir as tarefas do ligador e do compilador.

O compilador gerava o código binário executável onde os endereços de todos os símbolos e rótulos eram relativos ao endereço zero. Este código binário é a entrada do ligador, que percorria o código binário buscando as informações que deviam ser armazenadas no cabeçalho .EXE (Algoritmo 9.1). Algumas informações como o tamanho do arquivo são bem fáceis de calcular, mas outras são mais difíceis.

A mais complexa destas tarefas era gerar a lista de *fixups*. Para tal, após a criação do arquivo binário do código executável cria-se uma lista contendo a localização de todas as referências a símbolos contidos naquele arquivo binário. Por exemplo, ao encontrar a instrução `JUMP 0x1234`, a lista não continha o endereço do código binário do `JUMP`, mas sim do parâmetro dele, `0x1234`.

Ao chegar ao fim do arquivo binário contendo o programa executável, a lista de *fixups* estava completa. Aliás, como o cabeçalho foi sendo preenchido ao longo da leitura, o cabeçalho já estava quase completo também. O passo final era escrever o arquivo executável. Para isso, bastava escrever o cabeçalho e em seguida todo o arquivo binário do código executável em um arquivo com extensão .EXE.

Posteriormente, tanto o CP/M quanto o MS-DOS desenvolveram arquivos objeto (bibliotecas) e as respectivas ferramentas. Não encontrei referências conclusivas, mas aparentemente os formatos (e consequentemente as ferramentas) não são compatíveis.

Para mais informações sobre o funcionamento dos ligadores .EXE, veja [17].

9.2.3 Ligadores com seções

À medida que o desenvolvimento de sistemas operacionais foi evoluindo, constatou-se que o uso de arquivos objeto era benéfico por vários motivos, em especial para evitar o custo de recompilar código utilizado em vários programas. Por esta razão, foram criados formatos específicos para arquivos objeto.

⁷ Integrated Development Environment (IDE)

⁸ Era possível utilizar o compilador e o ligador em linha de comando (como também executar os programas desenvolvidos), porém não conheci muita gente que fazia isso.

Assim, o ligador passou a gerar arquivos objeto além dos executáveis. Nos arquivos objeto ele passou a incluir informações que o auxiliariam na geração dos arquivos executáveis. Nos arquivos executáveis ele passou a incluir informações que auxiliariam nas tarefas do programa carregador. O problema era como desenvolver um mecanismo que permitisse a inclusão destas informações nos dois tipos de arquivos de uma maneira simples.

Uma ideia natural é dividir os arquivos objeto e executável em pedaços com semântica semelhante. Por exemplo, um pedaço poderia conter informações sobre o código binário, outra sobre dados globais e assim por diante.

Exemplos desta abordagem incluem os formatos COFF, PE e ELF. Como o formato ELF é adotado no linux e tem documentação pública, esta seção aborda somente este formato.

A primeira coisa a destacar é que nos arquivos ELF objeto, os pedaços são chamados seções e nos arquivos ELF executável eles são chamados segmentos (que por sua vez também contêm seções).

Algumas seções tem nome e semântica pré-definidas (como por exemplo seção `.text` e `.data`), mas que é possível acrescentar seções e segmentos permitindo que o formato possa ser ajustado às mais diversas necessidades e interesses (veja Exercício 9.4).

Conforme explicado no Capítulo 8, os arquivos ELF podem estar em dois formatos: o formato ELF executável (Seção 8.1.2) e o formato ELF objeto (Seção 8.1.1). Desta forma, o programa ligador ELF recebe como entrada um ou mais arquivos ELF objeto e gera como saída um arquivo ELF executável.

Resumidamente, o ligador pode ser explicado com duas fases⁹:

Passo 1: Coleta de informações:

1. lê os arquivos de entrada para saber quais seções estão presentes assim como o seu tamanho. Com esta informação, ele é capaz de definir quantos e quais serão os segmentos do arquivo de saída, assim como o tamanho de cada um e as seções que cada segmento terá.
2. Cada arquivo ELF objeto contém uma seção com a tabela de símbolos. Alguns são símbolos importados (por exemplo, chamadas a procedimentos que não estão implementadas naquele arquivo objeto) e outros são os símbolos exportados (por exemplo, funções que outros arquivos objeto podem referenciar). O ligador lê estas tabelas e cria uma tabela de símbolos para o arquivo de saída. Nesta tabela de saída são indicadas as localizações (*offsets*) para cada símbolo em cada arquivo objeto.

Passo 2: Relocação e geração do arquivo de saída a partir das informações coletadas no primeiro passo:

1. Após mapear as seções em segmentos, calcula os endereços efetivos dos símbolos e executa a relocação dos mesmos.
2. insere cabeçalho, tabela de símbolos entre outros ajustes e grava o arquivo de saída.

Para ilustrar o funcionamento do ligador GNU na prática, a Seção 9.2.4 apresenta um conjunto de programas escritos em linguagem C, os respectivos arquivos objeto e o arquivo executável.

As demais seções abordam alguns aspectos do ligador em termos genéricos e utilizam os programas da Seção 9.2.4 para ilustrar como o ligador converte os arquivos objeto em executável.

A Seção 9.2.5 descreve como mapear as seções em segmentos e a Seção 9.2.7 descreve como funciona a relocação.

9.2.4 Exemplo Prático

Para ilustrar o funcionamento do ligador, utilizaremos um exemplo prático onde quatro programas fonte escritos em linguagem C (`01.c`, `02.c`, `03.c` e `04.c`) são compilados e geram quatro arquivos objeto (`01.o`, `02.o`, `03.o` e `04.o`).

O arquivo `01.c` é apresentado no Algoritmo 9.3. Ele faz chamadas aos procedimentos `P2()`, `P3()` e `P4()` que estão implementados nos arquivos `02.c` (Algoritmo 9.4), `03.c` (Algoritmo 9.5) e `04.c` (Algoritmo 9.6) respectivamente. Ele também utiliza quatro variáveis globais, `G1`, `G2`, `G3` e `G4`, sendo que `G1` é declarado no próprio arquivo `01.c` enquanto que as demais estão declarados nos arquivos `02.c`, `03.c` e `04.c` respectivamente. A diretiva `extern` da linha 8 diz ao compilador que as variáveis listadas serão encontradas em outro arquivo objeto.

9.2.5 Alocação de Espaço

Esta seção explica como é feita a alocação de espaço nos arquivos executáveis, em especial o mapeamento de seções em segmentos.

⁹ Na prática, ele tem mais fases conforme descrito em [44]

```

1  #include <stdio.h>
2
3  int P2 ();    // prototipos de funcoes declaradas em outros arquivos
4  int P3 ();
5  int P4 ();
6
7  int G1;      // global declarada neste arquivo
8  extern int G2, G3, G4; // globais declaradas em outros arquivos
9
10 int main () {
11     G1=15;
12     P2(); G2=2;
13     P3(); G3=3;
14     P4(); G4=4;
15 }

```

Algoritmo 9.3 – Arquivo 01.c

```

1  int G2;
2
3  int P2 () {
4     return(1);
5 }

```

Algoritmo 9.4 – Arquivo 02.c

```

1  int G3;
2
3  int P3 () {
4     return(1);
5 }

```

Algoritmo 9.5 – Arquivo 03.c

Considere vários arquivos de entrada para o ligador, digamos $0_1, 0_2 \dots 0_n$. Cada arquivo objeto está organizado em “pedaços” com semântica equivalente. Por exemplo, um “pedaço” contém o código binário executável (seção `.text`), outro “pedaço” contém informações sobre as variáveis globais (seção `.data`) e assim por diante.

Uma das tarefas do ligador é concatenar os “pedaços” equivalentes dos vários arquivos objeto em um único “pedaço” no arquivo executável.

Como exemplo, considere uma situação simples com quatro arquivos objeto onde cada um é composto por um único “pedaço”. Os tamanhos dos arquivos estão indicados na Tabela 16.

Arquivo	Tamanho
01.o	0x1017
02.o	0x920
03.o	0x615
04.o	0x1390

Tabela 16 – Arquivos objeto e seus tamanhos

Ao criar o arquivo executável, o ligador deve concatenar os arquivos objeto. Existem 24 (4!) formas diferentes de concatená-los, uma delas indicada na Figura 25. O lado esquerdo da figura mostra cada arquivo objeto isoladamente e seus respectivos *offsets*. Como mostra a figura, cada arquivo objeto começa no *offset* zero até o *offset* $T - 1$ onde T é o tamanho do arquivo objeto.

O centro da figura mostra o arquivo executável com endereços indicados a partir do início do arquivo. No lado direito da figura todos os *offsets* foram acrescidos de 0x1000. Esta situação representa os endereços que o programa

```

1 int G4;
2
3 int P4 () {
4     return(1);
5 }

```

Algoritmo 9.6 – Arquivo 04.c

executável teria se o arquivo fosse copiado na memória a partir do endereço físico 0x1000.

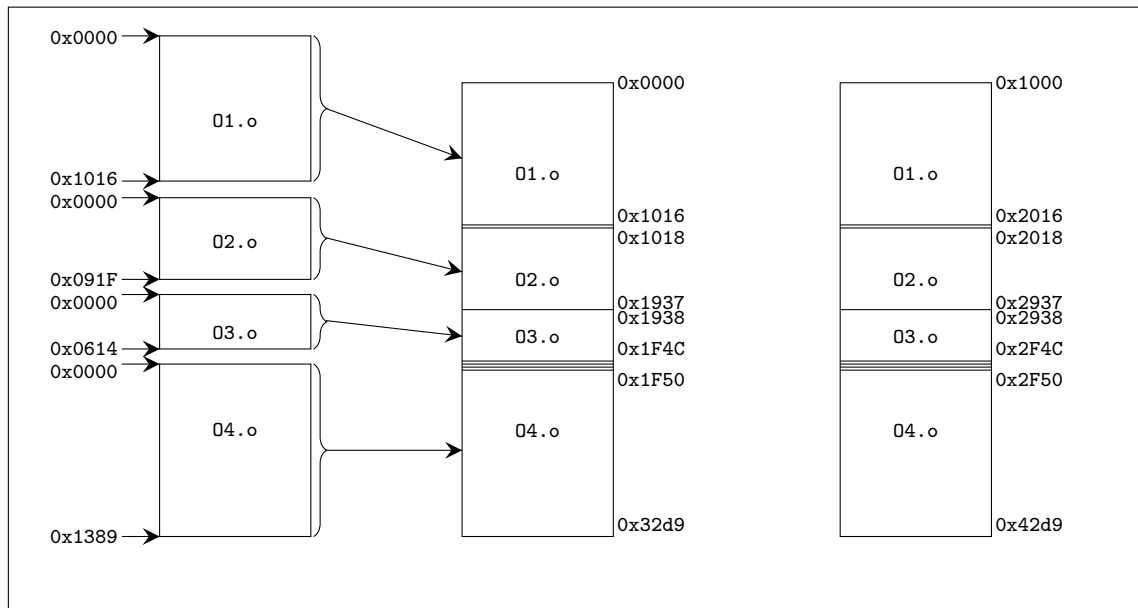


Figura 25 – Concatenação de arquivos objeto com uma única seção (esquerda) para criar um arquivo executável não relocado (centro) e relocado (direita).

Uma observação importante é que o ligador pode deixar alguns espaços entre os arquivos objeto que compõem o executável. No exemplo, foi deixado um byte entre os arquivos 01.o e 02.o (*offset* 0x1017) e três bytes entre os arquivos 03.o e 04.o (*offsets* 0x1f4d, 0x1f4e, 0x1f4f).

Neste exemplo, este deslocamento teve o objetivo de iniciar cada arquivo em um endereço múltiplo de quatro: no arquivo 01.o começa no *offset* 0x0000, no arquivo 02.o começa no *offset* 0x1018 e no arquivo 03.o começa no *offset* 0x1938.

Numa situação real, o objetivo poderia ser iniciar cada “pedaço” em uma página diferente de memória virtual.

A Figura 26 estende o exemplo da Figura 25 para a situação onde cada arquivo objeto tem dois “pedaços”, que são genericamente representados por duas cores. Evidentemente o arquivo executável também terá duas cores.

- nem todos os “pedaços” dos arquivos objeto serão mapeados no arquivo executável. Isto ocorre porque alguns “pedaços” contêm informações que são úteis para o ligador gerar o executável, porém inúteis para o carregador colocar o programa em execução.
- o arquivo executável pode ser acrescido de “pedaços” não existentes nos arquivos objeto. Estes “pedaços” contêm informações úteis ao carregador, mas que seriam inúteis para o ligador.

9.2.5.1 Alocação de Espaço no ELF

Para finalizar o tópico, esta seção mostra como é feita a alocação de espaço no ELF. São usados programas apresentados nos Algoritmos 9.3, 9.4, 9.5 E 9.6 (Seção 9.2.4).

Os comandos abaixo geram os arquivos objeto e executável:

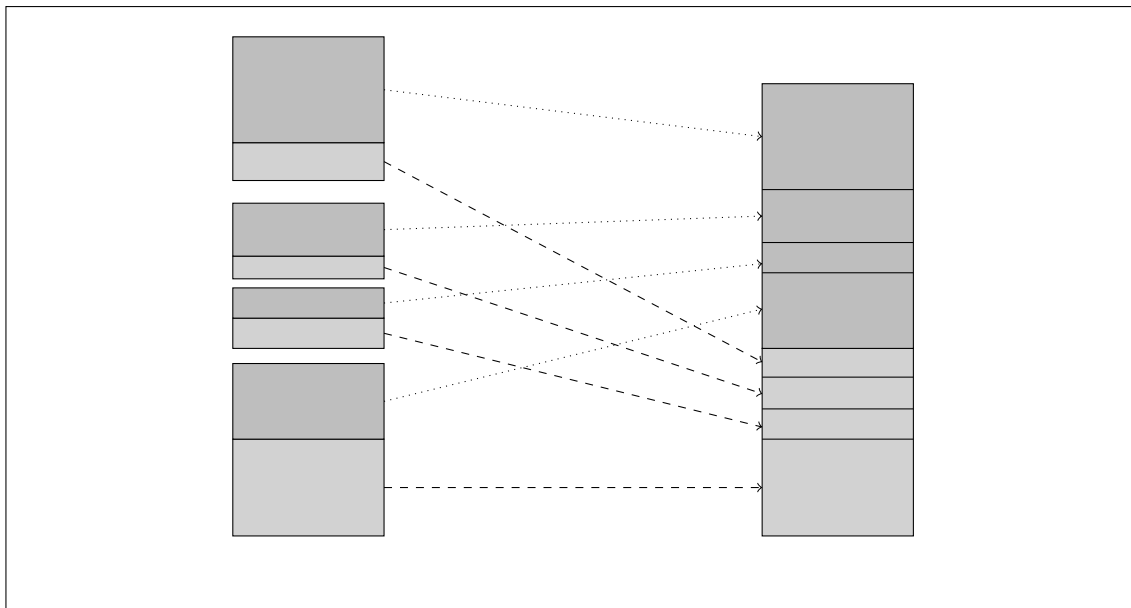


Figura 26 – Concatenação de arquivos objeto com dois “pedaços” cada

```
> gcc 02.c -c -o 02.o
> gcc 03.c -c -o 03.o
> gcc 04.c -c -o 04.o
> gcc 01.c -c -o 01.o
> gcc -static 01.o 02.o 03.o 04.o -o Exec
```

Para visualizar o conteúdo dos arquivos objeto e executável, será utilizado o programa `readelf`. Primeiramente, vamos examinar o arquivo `02.o`.

```
> readelf 02.o --sections --wide
```

There are 11 section headers, starting at offset 0x230:

Section Headers:

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	0000000000000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	0000000000000000	000040	00000b	00	AX	0	0	1
[2]	.data	PROGBITS	0000000000000000	00004b	000000	00	WA	0	0	1
[3]	.bss	NOBITS	0000000000000000	00004b	000000	00	WA	0	0	1
[4]	.comment	PROGBITS	0000000000000000	00004b	000036	01	MS	0	0	1
[5]	.note.GNU-stack	PROGBITS	0000000000000000	000081	000000	00		0	0	1
[6]	.eh_frame	PROGBITS	0000000000000000	000088	000038	00	A	0	0	8
[7]	.rela.eh_frame	RELA	0000000000000000	0001c0	000018	18	I	9	6	8
[8]	.shstrtab	STRTAB	0000000000000000	0001d8	000054	00		0	0	1
[9]	.symtab	SYMTAB	0000000000000000	0000c0	0000f0	18		10	8	8
[10]	.strtab	STRTAB	0000000000000000	0001b0	00000c	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
 I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
 0 (extra OS processing required) o (OS specific), p (processor specific)

Este arquivo tem 11 “pedaços”, que o ELF chama de seções (veja Figura 20 na página 108). As colunas indicam as informações contidas em cada seção. Cada coluna indica a informação contida no campo da estrutura ELF correspondente (veja Algoritmo 8.7 da página 112).

Nr Número da seção;

Name (sh_name) Nome da seção. Alguns já conhecidos como .text, .data, .bss.

Type (sh_type) Indica a semântica da seção.

Address (sh_name) Endereço virtual em tempo de execução.

Off (sh_offset) *Offset* onde inicia a seção no arquivo objeto.

Size (sh_name) O tamanho (em número de bytes) daquela seção.

ES (sh_entsize) *Entry Size*

Flg (sh_flags) Estão indicados no final da listagem. Por exemplo, a seção .text tem flags AX, indicando que ocupa espaço em tempo de execução, e que é uma seção que contém código executável.

Lk (sh_link) Indica se está relacionada com outra seção. No exemplo, a seção 7 está relacionada com a 9 que está relacionada com a 10.

Inf (sh_info) Informação adicional.

Al (sh_addralign) Alinhamento da seção.

Omitimos a listagem dos demais arquivos objeto (01.o, 03.o, e 04.o) pois as informações são semelhantes já que os arquivos-fonte são semelhantes.

O ligador recebe estes quatro arquivos objeto e gera o arquivo executável Exec listado abaixo.

```
> readelf Exec --sections --wide
```

There are 31 section headers, starting at offset 0x1ab8:

Section Headers:

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	0000000000000000	000000	000000	00		0	0	0
[1]	.interp	PROGBITS	0000000000400238	000238	00001c	00	A	0	0	1
[2]	.note.ABI-tag	NOTE	0000000000400254	000254	000020	00	A	0	0	4
[3]	.note.gnu.build-id	NOTE	0000000000400274	000274	000024	00	A	0	0	4
[4]	.gnu.hash	GNU_HASH	0000000000400298	000298	00001c	00	A	5	0	8
[5]	.dynsym	DYNSYM	00000000004002b8	0002b8	000048	18	A	6	1	8
[6]	.dynstr	STRTAB	0000000000400300	000300	000038	00	A	0	0	1
[7]	.gnu.version	VERSYM	0000000000400338	000338	000006	02	A	5	0	2
[8]	.gnu.version_r	VERNEED	0000000000400340	000340	000020	00	A	6	1	8
[9]	.rela.dyn	RELA	0000000000400360	000360	000018	18	A	5	0	8
[10]	.rela.plt	RELA	0000000000400378	000378	000018	18	AI	5	24	8
[11]	.init	PROGBITS	0000000000400390	000390	00001a	00	AX	0	0	4
[12]	.plt	PROGBITS	00000000004003b0	0003b0	000020	10	AX	0	0	16
[13]	.plt.got	PROGBITS	00000000004003d0	0003d0	000008	00	AX	0	0	8
[14]	.text	PROGBITS	00000000004003e0	0003e0	0001e2	00	AX	0	0	16
[15]	.fini	PROGBITS	00000000004005c4	0005c4	000009	00	AX	0	0	4
[16]	.rodata	PROGBITS	00000000004005d0	0005d0	000004	04	AM	0	0	4
[17]	.eh_frame_hdr	PROGBITS	00000000004005d4	0005d4	00004c	00	A	0	0	4
[18]	.eh_frame	PROGBITS	0000000000400620	000620	000154	00	A	0	0	8
[19]	.init_array	INIT_ARRAY	0000000000600e10	000e10	000008	00	WA	0	0	8
[20]	.fini_array	FINI_ARRAY	0000000000600e18	000e18	000008	00	WA	0	0	8
[21]	.jcr	PROGBITS	0000000000600e20	000e20	000008	00	WA	0	0	8
[22]	.dynamic	DYNAMIC	0000000000600e28	000e28	0001d0	10	WA	6	0	8
[23]	.got	PROGBITS	0000000000600ff8	000ff8	000008	08	WA	0	0	8
[24]	.got.plt	PROGBITS	0000000000601000	001000	000020	08	WA	0	0	8
[25]	.data	PROGBITS	0000000000601020	001020	000010	00	WA	0	0	8
[26]	.bss	NOBITS	0000000000601030	001030	000018	00	WA	0	0	4
[27]	.comment	PROGBITS	0000000000000000	001030	000035	01	MS	0	0	1
[28]	.shstrtab	STRTAB	0000000000000000	0019ab	00010c	00		0	0	1
[29]	.symtab	SYMTAB	0000000000000000	001068	000720	18		30	50	8
[30]	.strtab	STRTAB	0000000000000000	001788	000223	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
 I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
 O (extra OS processing required) o (OS specific), p (processor specific)

Destaques:

- no arquivo executável ELF, os “pedaços” são chamados de segmentos e não de seções. Apesar disto, os nomes são usados indistintamente (veja a primeira linha da listagem).
- A listagem indica a existência de 30 segmentos, e as colunas mostram os mesmos atributos contidos nos arquivos objeto.
- A coluna Address indica o endereço virtual onde o carregador deve copiar aquele segmento.
- O ligador usou as sugestões indicadas no campo *flags* e copiou no atributo correspondente do executável.

9.2.6 Tabela de Símbolos

Em formatos mais modernos, cada arquivo objeto contém uma tabela de símbolos. O ligador lê as tabelas de símbolos de cada arquivo objeto e cria uma tabela unificada onde pode, por exemplo, verificar se todos os símbolos necessários estão presentes em algum arquivo. Em seguida, esta tabela é incrementada com informações para relocação.

Existem várias formas de armazenar as tabelas de símbolos, e várias formas de implementar a tabela unificada. Veja [25] para mais informações.

Vamos nos concentrar na tabela de símbolos do ELF. Para tal, considere os programas da Seção 9.2.4.

A tabela de símbolos de cada arquivo objeto ELF é armazenada na seção `.symtab`. O ligador lê as tabelas de símbolos de cada arquivo de entrada para uma tabela única utilizada para gerar o executável. Esta tabela é complementada com informações de outras seções, por exemplo, para fazer a relocação. Porém, antes de complementá-la, é possível descobrir se cada símbolo necessário está presente. Vamos examinar as tabelas de símbolos dos arquivos `02.o` e `01.o` utilizando o comando `readelf -s` que lista a seção `.symtab`.

```
> readelf 02.o -s
```

Symbol table '.symtab' contains 10 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	02.c
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	2	
4:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	
5:	0000000000000000	0	SECTION	LOCAL	DEFAULT	5	
6:	0000000000000000	0	SECTION	LOCAL	DEFAULT	6	
7:	0000000000000000	0	SECTION	LOCAL	DEFAULT	4	
8:	0000000000000004	4	OBJECT	GLOBAL	DEFAULT	COM	G2
9:	0000000000000000	11	FUNC	GLOBAL	DEFAULT	1 P2	

O ELF também considera que seções são símbolos e que o próprio nome do arquivo é um símbolo. Porém o que interessa aqui são as duas últimas linhas, onde os símbolos G2 e P2 são indicados como objeto global e função global respectivamente. Em outras palavras, são símbolos exportados por este arquivo. Confira o código fonte deste arquivo para ver que ele não requer símbolos externos.

Já o arquivo `01.o` utiliza símbolos não definidos nele:

```
> readelf 01.o -s
```

Symbol table '.symtab' contains 16 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	01.c
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	
4:	0000000000000000	0	SECTION	LOCAL	DEFAULT	4	

```

5: 0000000000000000    0 SECTION LOCAL DEFAULT    6
6: 0000000000000000    0 SECTION LOCAL DEFAULT    7
7: 0000000000000000    0 SECTION LOCAL DEFAULT    5
8: 0000000000000004    4 OBJECT GLOBAL DEFAULT COM G1
9: 0000000000000000   81 FUNC GLOBAL DEFAULT    1 main
10: 0000000000000000    0 NOTYPE GLOBAL DEFAULT UND P2
11: 0000000000000000    0 NOTYPE GLOBAL DEFAULT UND G2
12: 0000000000000000    0 NOTYPE GLOBAL DEFAULT UND P3
13: 0000000000000000    0 NOTYPE GLOBAL DEFAULT UND G3
14: 0000000000000000    0 NOTYPE GLOBAL DEFAULT UND P4
15: 0000000000000000    0 NOTYPE GLOBAL DEFAULT UND G4

```

Observe as últimas linhas. Os símbolos G1 e main são definidos no arquivo (exportados) enquanto que os símbolos P2, G2, P3, G3, P4 e G4 não são definidos (UND na penúltima coluna), e portanto devem ser disponibilizados por algum outro arquivo objeto.

9.2.7 Relocação

Os endereços indicados nos arquivos objeto são relativos ao início do mesmo arquivo (são *offsets*). Exemplos de endereços incluem os rótulos de desvio e os símbolos globais (variáveis e rótulos).

Quando os arquivos objeto são agrupados em um arquivo executável, estes endereços devem ser ajustados ou para o *offset* do arquivo executável ou para os endereços efetivos em tempo de execução.

O processo é simples:

1. o ligador cria uma tabela contendo todos os rótulos e símbolos contidos nos arquivos objeto que devem ser relocados e os locais onde são referenciados. A tabela também indica quais símbolos são exportados e quais são importados em cada arquivo objeto. Todos os símbolos referenciados devem aparecer na tabela exatamente uma vez;
2. o ligador concatena os arquivos objeto atualizando a tabela com os *offsets* do arquivo executável;
3. por fim, o ligador atualiza o arquivo executável substituindo os endereços indicados nos *offsets* com os endereços que serão utilizados em tempo de execução.

O processo descrito acima é basicamente o mesmo em todos os ligadores, porém técnicas diferentes podem ser aplicadas dependendo das características do *hardware* (como a presença de registradores de segmento).

Se o formato executável incluir seções, técnicas particulares podem ser adotadas em cada seção. Por exemplo, na seção `.text` é interessante evitar instruções de desvio absoluto e utilizar instruções de desvio relativas. A diferença entre as duas está indicada na Tabela 17. A primeira linha mostra a instrução de desvio para endereço absoluto, onde o parâmetro é copiado para o `%rip`. A segunda linha mostra uma instrução de desvio relativo, onde o parâmetro (em complemento de dois) é somado ao registrador `%rip`.

Instrução	Descrição	Ação
<code>jmp <end></code>	<i>JMP Far</i>	<code>%rip ← <end></code>
<code>jmp <desl></code>	<i>JMP Near</i>	<code>%rip ← %rip + <desl></code>

Tabela 17 – Instruções de desvio absoluto (linha 1) e relativo (linha 2) no AMD64

A segunda coluna mostra como cada tipo de instrução é chamado no AMD64. Os termos *Far* e *Near* são utilizados para indicar endereços fora e dentro do segmento de código respectivamente.

Uma outra técnica comum é utilizada para acessar as variáveis globais a partir do endereço da instrução corrente. Quando um programa é colocado em execução, o endereço das variáveis globais é fixo, assim como o endereço das instruções. Compiladores e ligadores sabem quais são estes endereços, e como sabem a distância (em bytes) entre a instrução atual e o endereço de uma variável global, podem utilizar instruções como a descrita abaixo (sintaxe AMD64).

```
movq $14, 0x200b41(%rip)
```

Esta instrução copia a constante 14 para uma variável global cujo endereço é `0x200b41 + %rip`. Neste caso, o arquivo objeto terá uma instrução como `movq $14, 0x0(%rip)`, e `0x0` é o *fixup*. Esta técnica é muito utilizada para carregar bibliotecas dinâmicas como será explicado na Capítulo 10.

9.2.8 Relocação no ELF

Vamos agora examinar como as informações de relocação são armazenadas no formato ELF. Para tal, serão utilizados os programas dos Algoritmos 9.3, 9.4, 9.5 E 9.6. Após compilá-los e gerar os respectivos arquivos objeto e executável, usaremos o programa `readelf` para visualizar as seções e `objdump` para examinar o código.

O comando `readelf -r` permite visualizar as seções de relocação. Vamos iniciar com o arquivo `02.o`:

```
> readelf 02.o -r
```

Relocation section '.rela.eh_frame' at offset 0x4c8 contains 1 entries:

Offset	Info	Type	Sym. Value	Sym. Name + Addend
0000000000020	0002000000002	R_X86_64_PC32	0000000000000000	.text + 0

Esta seção indica que a seção `.rela.eh_frame` contém uma entrada a ser relocada, no *offset* 0x20. Indica também que esta entrada é do tipo `R_X86_64_PC32`, ou seja, que é uma entrada cujo endereço é relativo ao PC com deslocamento ocupando 32 bits¹⁰.

Esta entrada corresponde à variável global `G2`, e está referenciada na seção `.rela.eh_frame` porque lá é o local para armazenar informações de depuração (veja Exercícios 9.8 E 9.9).

O arquivo objeto `01.o` contém mais informações a serem relocadas:

```
> readelf 01.o -r
```

Relocation section '.rela.text' at offset 0x5f0 contains 7 entries:

Offset	Info	Type	Sym. Value	Sym. Name + Addend
0000000000006	0008000000002	R_X86_64_PC32	0000000000000004	G1 - 8
0000000000014	000a000000002	R_X86_64_PC32	0000000000000000	P2 - 4
000000000001a	000b000000002	R_X86_64_PC32	0000000000000000	G2 - 8
0000000000028	000c000000002	R_X86_64_PC32	0000000000000000	P3 - 4
000000000002e	000d000000002	R_X86_64_PC32	0000000000000000	G3 - 8
000000000003c	000e000000002	R_X86_64_PC32	0000000000000000	P4 - 4
0000000000042	000f000000002	R_X86_64_PC32	0000000000000000	G4 - 8

Relocation section '.rela.eh_frame' at offset 0x698 contains 1 entries:

Offset	Info	Type	Sym. Value	Sym. Name + Addend
0000000000020	0002000000002	R_X86_64_PC32	0000000000000000	.text + 0

A primeira coluna indica o offset **dentro da seção** `.text` (daí o nome da seção ser `.rela.text`). Vamos analisar o conteúdo da seção `.text`:

```
> objdump 01.o -S
```

```
01.o:      file format elf64-x86-64
```

Disassembly of section `.text`:

```
0000000000000000 <main>:
```

```

0:      55                      push  %rbp
1:      48 89 e5                mov   %rsp,%rbp
4:      c7 05 00 00 00 00 0f  movl  $0xf,0x0(%rip)      # e <main+0xe>
b:      00 00 00
e:      b8 00 00 00 00        mov   $0x0,%eax
13:     e8 00 00 00 00        callq 18 <main+0x18>
18:     c7 05 00 00 00 00 02  movl  $0x2,0x0(%rip)      # 22 <main+0x22>
1f:     00 00 00
22:     b8 00 00 00 00        mov   $0x0,%eax
27:     e8 00 00 00 00        callq 2c <main+0x2c>
```

¹⁰ indicado no arquivo `/usr/include/elf.h`: PC relative 32 bit signed

```

2c:      c7 05 00 00 00 00 03      movl    $0x3,0x0(%rip)      # 36 <main+0x36>
33:      00 00 00
36:      b8 00 00 00 00      mov     $0x0,%eax
3b:      e8 00 00 00 00      callq   40 <main+0x40>
40:      c7 05 00 00 00 00 04      movl    $0x4,0x0(%rip)      # 4a <main+0x4a>
47:      00 00 00
4a:      5d                      pop     %rbp
4b:      c3                      retq

```

A primeira relocação é um `R_X86_64_PC32` (ou seja, 32 bits numa instrução relativa ao PC) no *offset* 0x6, que está sublinhado abaixo.

```

4:      c7 05 00 00 00 00 0f      movl    $0xf, 0x0(%rip)

```

Para relocar este endereço, o ligador deve substituir os 32 bits sublinhados por outro valor. Isto altera a instrução à direita, pois mudará o 0x0 por outro valor.

É possível ver esta alteração no arquivo `Exec`. A impressão é muito longa (205 linhas), incluindo código não presente nos arquivos de entrada.

Para gerar o arquivo `Exec`, o ligador considera que o endereço de entrada no código (rótulo `_start`) é o endereço 0x400400. Outros trechos de código são colocados acima dele fazendo com que o primeiro endereço da seção `.text` seja 0x4003a8.

Os demais trechos de código são colocados abaixo de `_start`, como por exemplo, o rótulo `main` no endereço 0x4004ed.

Após preencher a seção `.text`, o ligador preenche a seção `.data` estipulando um endereço de início para esta seção em tempo de execução. Como o endereço das variáveis globais é fixo, o ligador sabe qual a distância entre cada instrução e a variável global.

Neste caso específico, o ligador poderia utilizar o endereçamento direto para acessar a variável global, mas como utilizou o endereçamento indireto, o *fixup* que citamos acima foi ajustado para 0x200b41(%rip). O comentário à direita indica que a variável acessada é `G1` no endereço 0x60103c, pois 0x200b41 + 0x4004fb = 0x60103c.

```

...
0000000004004ed <main>:
4004ed:      55                      push    %rbp
4004ee:      48 89 e5                mov     %rsp,%rbp
4004f1:      c7 05 41 0b 20 00 0f      movl    $0xf,0x200b41(%rip) # 60103c <G1>
4004f8:      00 00 00
4004fb:      b8 00 00 00 00      mov     $0x0,%eax
400500:      e8 34 00 00 00      callq   400539 <P2>
400505:      c7 05 31 0b 20 00 02      movl    $0x2,0x200b31(%rip) # 6010
...

```

Se o leitor observar cuidadosamente o código gerado pelo ligador, perceberá muito código externo (que não está no programa `C`, como as seções `_init`, `.plt`, `deregister_tm_clones`, `frame_dummy`, `_libc_csu_init`, `_frame_dummy_init_array_entry`, `_libc_csu_fini` e `fini`).

Alguns deles são incluídos pelos arquivos `/usr/lib/x86_64-linux-gnu/crtn.o`, `/usr/lib/x86_64-linux-gnu/crti.o` e `/usr/lib/x86_64-linux-gnu/crti.o`:

crtn.o Contém o rótulo `_start` que inicia a construção do registro de ativação com as variáveis `argc` e `argv`.

crti.o Contém a função `_init` na seção `.init`. Esta seção contém instruções a serem executadas após o carregamento do programa, mas antes de iniciar a execução (antes de chegar em `main`¹¹).

crtn.o Contém a função `_fini` na seção `.fini`. Esta seção contém instruções a serem executadas quando o programa for finalizado (após sair de `main`). Para mais informações, consulte <https://gcc.gnu.org/onlinedocs/gccint/Initialization.html>.

Exercícios

- 9.1 Na estrutura do cabeçalho do formato `.EXE` (Algoritmo 9.1), por que é necessário indicar o endereço de início da tabela `*relocs[]` na variável `relocpos`?

¹¹ http://14u-00.jinr.ru/usoft/WWW/www_debian.org/Documentation/elf/node3.html

- 9.2 O modelo de relocação apresentado para o formato .EXE não levou em consideração o endereçamento segmentado. Considere que o ligador consegue dividir o executável em pelo menos três segmentos: dados, código e pilha e que todos os acessos à memória utilizam a notação `Segm:Desloc` (onde o segmento é dado pelos registradores DS, CS e SS - dados, código e pilha respectivamente). Descreva como o carregador .EXE deve ser modificado para tratar com isto.
- 9.3 O modelo .EXE também permitia o uso de overlays (Seção D.1). Descreva que informações o ligador deve acrescentar ao cabeçalho e como o carregador os interpreta para colocar o programa em execução.
- 9.4 Considere incluir um segmento no arquivo executável ELF que contém a assinatura digital do aplicativo. A ideia é que o carregador confira a assinatura digital do arquivo com o fornecido pela empresa desenvolvedora do aplicativo e possa com isso garantir a autenticidade do produto. Que alterações devem ser feitas no carregador? Para uma descrição completa, veja [12].
- 9.5 Examine (`objdump -S`) os arquivos objeto (`0?.o`) e o executável (`Exec`). Como o ligador concatenou as várias seções `.text`?
- 9.6 Os ligadores precisam lidar com dependência circular em bibliotecas. Este tipo de dependência ocorre quando uma biblioteca, digamos AA precisa de um símbolo exportado por BB, que por sua vez precisa de um símbolo exportado por AA. Como exemplo, considere os programas fonte `func_dep.c` (Algoritmo 9.7) e `bar_dep.c` (Algoritmo 9.8). Suponha que estes são utilizados no programa `simplemain.c` (Algoritmo 9.9). Após criar as bibliotecas e tentar executar o programa, ocorre o problema indicado abaixo. Explique por que ocorre este problema, e por que foi solucionado com o método alternativo apresentado por último. Sugestão: desenhe um grafo de dependências.

```
> ar -cr bar_dep.a bar_dep.o
> ar -cr func_dep.a func_dep.o
> gcc simplemain.c -c -o simplemain.o
> gcc simplemain.o -L. -lbar_dep -lfunc_dep
./libfunc_dep.a(func_dep.o): In function 'func':
func_dep.c:(.text+0x14): undefined reference to 'bar'
collect2: error: ld returned 1 exit status
> gcc simplemain.o -L. -lfunc_dep -lbar_dep
> ./a.out ; echo $?
4
>
```

Este problema pode ficar mais complexo e a solução acima pode não funcionar, e é necessário utilizar as diretivas `-start-group` e `-end-group`¹²

```
1 int bar(int);
2
3 int func(int i) {
4     return bar(i + 1);
5 }
```

Algoritmo 9.7 – Arquivo `func_dep.c`

- 9.7 No exercício anterior, o problema ocorreu com bibliotecas. O mesmo problema ocorre com os arquivos objeto, por exemplo:

```
> gcc simplemain.o -L. bar_dep.o func_dep.o
> gcc simplemain.o -L. func_dep.o bar_dep.o
```

Isto implica dizer que o ligador trata os símbolos de bibliotecas e os símbolos de arquivos objeto de maneira igual?

- 9.8 No exemplo apresentado sobre a relocação no ELF (Seção 9.2.8) a variável global `G2` era referenciada na seção `rela.eh_frame`. Se for incluída uma atribuição, por exemplo, `G2=1` no procedimento `P2`, como ficam as informações sobre relocação no arquivo objeto?

¹² <http://eli.thegreenplace.net/2013/07/09/library-order-in-static-linking#circular-dependency>

```
1 int func(int);
2
3 int bar(int i) {
4     if (i > 3)
5         return i;
6     else
7         return func(i);
8 }
```

Algoritmo 9.8 – Arquivo bar_dep.c

```
1 int func(int);
2
3 int main(int argc, const char* argv[])
4 {
5     return func(argc);
6 }
```

Algoritmo 9.9 – Arquivo simplemain.c

9.9 O formato DWARF¹³, é a extensão do ELF¹⁴ que inclui informações sobre depuração. Compile os programas com "gcc ... -g", que gera arquivos objeto e executáveis no formato DWARF) e examine as informações sobre relocação (contém seções não discutidas aqui).

9.10 Execute a mão o trabalho que o ligador teve para gerar o arquivo executável. Procure elaborar um algoritmo que explica o funcionamento do ligador para os tópicos citados nesta seção.

¹³ *Debug With Attributed Record Format*, Depuração Através de Registros com Atributos

¹⁴ Não sei qual a relação dos computadores com o livro "Senhor dos Anéis", mas descobri na prática que só os computadores acham graça nos nomes serem ELF e DWARF. Os não computadores dão um sorriso amarelo. Bem, eu não posso dizer que não discordo deles! (Groucho Marx)

10. O Ligador Dinâmico

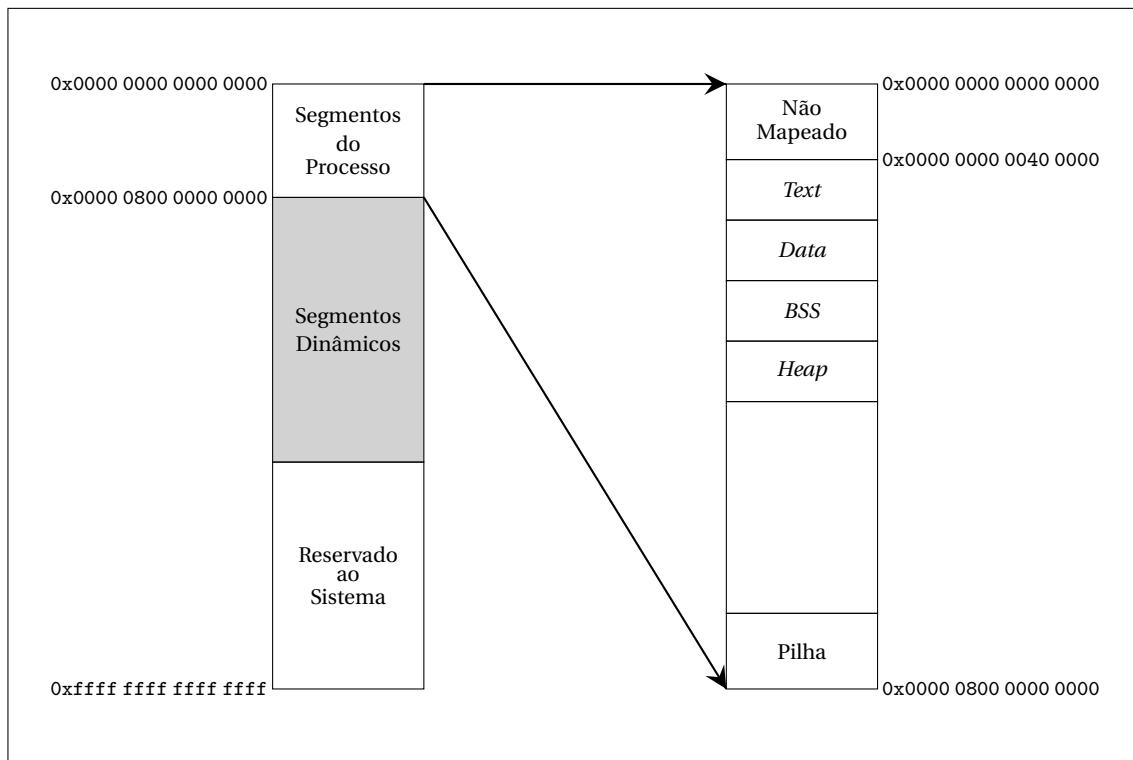


Figura 27 – Espaço Virtual de um Processo: Segmento Dinâmico (ou compartilhado)

O termo “ligador dinâmico” é utilizado em vários contextos, criando uma nomenclatura confusa. Por esta razão, vamos começar este capítulo padronizando a nomenclatura.

No mundo linux, o ligador dinâmico¹ é uma parte do sistema operacional que carrega e liga as bibliotecas compartilhadas² que um programa executável precisa para funcionar. Como é ele que carrega as bibliotecas compartilhadas, não é comum usar o termo carregador dinâmico³ que é usado como um sinônimo para ligador dinâmico.

O ligador em tempo de execução⁴ é uma parte do sistema operacional que carrega e liga as bibliotecas dinâmicas em tempo de execução.

Atenção: o ligador em tempo de execução lida com bibliotecas dinâmicas enquanto que o ligador dinâmico lida com bibliotecas compartilhadas. Fácil de confundir, mas fica pior: ambas utilizam a biblioteca `ld-linux-x86-64.so.2` denominada “ligador dinâmico”.

Para piorar mais ainda, há mais um nome que confunde: a “biblioteca dinâmica” `libdl` (`libdl.a` - versão estática - e `libdl.so` - versão compartilhada) apresentada na Seção 8.1.6.

Acho que com o que foi exposto acima, o leitor entendeu o porquê da expressão “nomenclatura confusa”. Por isso, é necessário muito cuidado com o que se encontra nos livros e sítios da internet.

Elaborei um resumo que pode ajudar no mundo linux:

¹ *dynamic linker*

² *shared libraries*

³ *dynamic loader*

⁴ *runtime linker*

- o ligador dinâmico do linux está na biblioteca `ld-linux-x86-64.so.2`. Ele é utilizado para carregar e ligar as bibliotecas compartilhadas (com extensão `.so`) que um programa executável precisa para funcionar. Ele é executado **única vez antes do programa iniciar a execução**;
- O ligador em tempo de execução carrega e liga bibliotecas compartilhadas (com extensão `.so`) **uma ou mais vezes ao longo da execução do um programa**. Para tal, utiliza a biblioteca dinâmica (`libdl`) que contém procedimentos como `dlopen()`, `dlsym()`, etc. (veja Seção 8.1.6). Estes procedimentos fazem uso de algumas das funcionalidades da biblioteca dinâmica `ld-linux-x86-64.so.2` para executar as tarefas requisitadas. Por exemplo, o procedimento `dlopen()` acessa as funcionalidades de `ld-linux-x86-64.so.2` para carregar uma biblioteca compartilhada na memória virtual.
- O ligador estático (`/usr/bin/ld`) visto no Capítulo 9 é responsável por gerar o arquivo executável. Ele faz parte de programas executado pelo `gcc` e por esta razão não é comum ser executado explicitamente. Neste livro só foi utilizado explicitamente na geração das bibliotecas compartilhadas `libMySharedLib.so` (página 117) e `libMyDynamic.so` (página 119).

Este capítulo concentra-se no ligador dinâmico, ou seja, aquele que é executado uma única vez antes de iniciar a execução de um programa. O ligador dinâmico carrega as bibliotecas necessárias na área da memória virtual do processo e liga os procedimentos e variáveis do programa principal com a biblioteca e vice-versa.

É importante ressaltar que este capítulo é, de longe, o mais complicado do livro. Isto ocorre porque o assunto é terrivelmente complicado, sendo que já o reescrevi várias vezes. Apesar de eu me esforçar muito para torná-lo mais estruturado, lógico e incremental, ainda me sinto frustrado com o resultado. Por outro lado, sinto-me menos frustrado quando leio qualquer outro material.

O capítulo está organizado em seções que podem ser vistas como “degraus”. Cada seção aprofunda o que foi apresentado nas seções anteriores, descendo mais um degrau.

A Seção 10.1 é o primeiro degrau, onde são apresentadas as vantagens do uso de bibliotecas compartilhadas e também dos desafios envolvidos para fazê-las funcionar.

Os problemas exigem soluções complicadas e envolvem instruções específicas do *hardware* que são apresentadas na Seção 10.2, assim como ideias de como utilizá-las na prática.

Por fim, a Seção 10.3 descreve como as instruções e ideias apresentadas na Seção 10.2 foram combinadas no ELF/Linux/AMD64 com um exemplo prático onde se faz análise de código objeto, código executável e execução (contém vários degraus).

10.1 Vantagens e Problemas

Bibliotecas compartilhadas e dinâmicas foram apresentadas nas Seções 8.1.5 e 8.1.6 respectivamente, onde foram destacadas algumas vantagens de seu uso em comparação com as bibliotecas estáticas.

Estas vantagens incluem:

- a criação de arquivos executáveis de tamanho menor. Isto ocorre pois não é incluído o código da biblioteca no arquivo executável, mas sim uma indicação do local onde elas podem ser encontradas;
- uso de menor quantidade de memória. Quando dois ou mais programas usam uma mesma biblioteca compartilhada, somente uma cópia da biblioteca é colocada na memória física;
- atualização mais eficiente de versão. Na biblioteca estática o código incluído no arquivo executável é aquele disponível na época da compilação. Se uma nova versão da biblioteca for disponibilizado (por exemplo, corrigindo um problema de segurança), ele só será incluído naquele arquivo executável se o programa for recompilado. No caso de uma biblioteca compartilhada, basta substituir o arquivo que contém a biblioteca compartilhada pela versão mais nova. Na próxima vez que o programa for colocado em execução, a nova versão será utilizada.

Porém, os problemas relacionados com dois ou mais processos compartilhando código e dados não foram abordados. Estes problemas podem ser vistos como problemas logísticos para o funcionamento destas bibliotecas.

A Figura 28 será utilizada para apresentar estes problemas. Lá, dois processos diferentes estão em execução. O processo 1 está mapeado na memória virtual e representado esquematicamente no lado esquerdo da figura enquanto o processo 2 está representado esquematicamente no lado direito da figura. Cada processo está dividido nas três regiões já conhecidas: segmento de processo na parte superior, segmento de objetos compartilhados ao centro (na cor cinza) e segmento do sistema na parte de baixo.

Os dois processos utilizam uma mesma biblioteca compartilhada, representado pela caixa branca na região de segmentos de objetos compartilhados. Como já foi destacado anteriormente, as bibliotecas podem ser uma mapeadas

em qualquer endereço virtual no segmento de objetos compartilhados, o que está sugerido na figura: no programa representado no lado esquerdo da figura, a biblioteca está mapeada em endereços maiores enquanto que no programa do lado direito da figura, está mapeada em endereços menores.

A parte central da figura mostra a biblioteca compartilhada mapeada na memória física. As linhas tracejadas indicam o mapeamento das páginas virtuais dos dois processos em páginas da memória física. Isto significa dizer que se o conteúdo de um determinado endereço de memória física da biblioteca compartilhada for alterado, essa alteração será visível instantaneamente no espaço virtual de endereçamento dos dois processos.

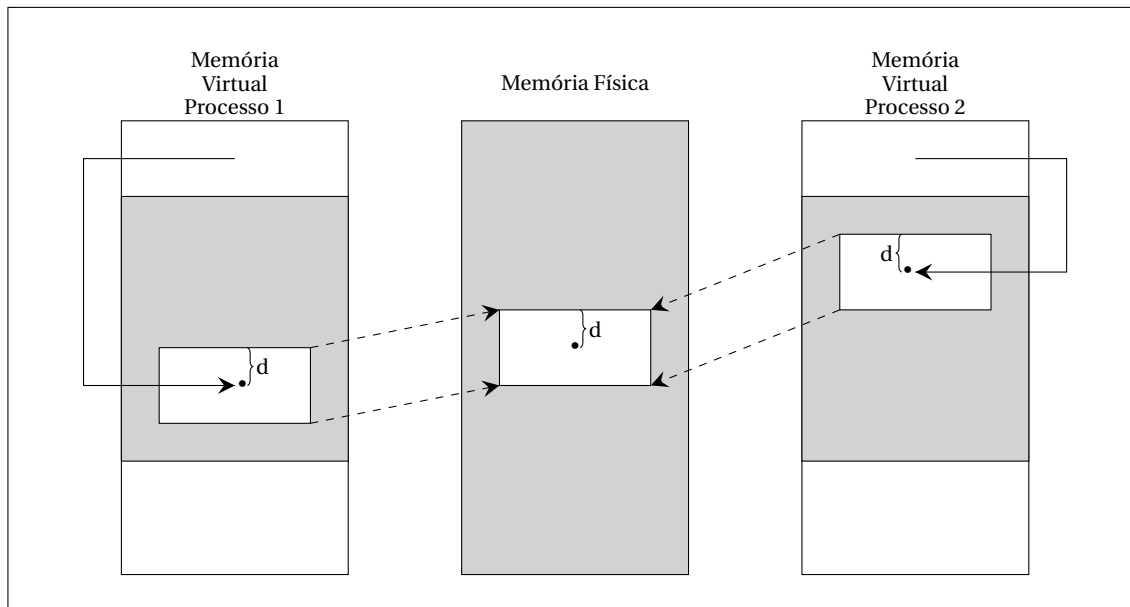


Figura 28 – Biblioteca compartilhada mapeada em dois processos (espaços virtuais de endereçamento diferentes).

Agora considere a chamada de um procedimento que reside na biblioteca compartilhada por exemplo o `printf`. Tanto o processo 1 quanto o processo 2 podem fazer várias chamadas a este procedimento, mas vamos nos concentrar em uma chamada de cada processo. A chamada do processo 1 está representada pela linha que sai da região de segmentos de processo e atinge o objeto compartilhado, e o mesmo ocorre no processo 2. Nos dois casos, o alvo é o rótulo (endereço) de início do procedimento `printf`.

Como os dois processos mapearam o mesmo objeto compartilhado em regiões diferentes de suas respectivas memórias virtuais, isto significa que o endereço alvo do procedimento é diferente em cada processo.

O problema é que as instruções de desvio apresentadas até o momento neste livro contém um endereço fixo como parâmetro (por exemplo, `jmp <endereço>` e `call <endereço>`), o que significa que elas não são apropriadas para esta situação.

Um outro aspecto importante a ser destacado é a ordem de carregamento. O carregador primeiro coloca na memória virtual as informações do segmento de processo, e só depois coloca na memória virtual a biblioteca compartilhada. Isto significa que o endereço dos procedimentos contidos no segmento de processo devem ser ajustados **APÓS** o carregamento da biblioteca compartilhada.

O problema com o “endereço alvo” das instruções de desvio também ocorre dentro do objeto compartilhado. Relembre que na Seção 2.3 foi apresentado o modelo de tradução do comando `while`, onde eram utilizados os dois rótulos: um para o início da repetição e outro para o fim da repetição. Como a biblioteca compartilhada pode residir em qualquer endereço virtual, as instruções de desvio para estes dois rótulos não podem utilizar endereços fixos. O mesmo problema ocorre quando um procedimento que reside no objeto compartilhado chamar um procedimento local ao objeto compartilhado. O endereço alvo das instruções de desvio para este procedimento local não pode ser fixo.

Agora vamos abordar um outro problema: o acesso às variáveis de um objeto compartilhado. Observe que o acesso às variáveis locais não é um problema, uma vez que elas são instanciadas na seção da pilha de cada processo e os endereços das variáveis lá contidas são calculados em função do registrador `%rbp` (por exemplo `-8(%rbp)`). De maneira análoga, as variáveis alocadas dinamicamente também não são um problema pois são alocadas na seção *heap* de cada processo.

O problema são as variáveis globais da biblioteca compartilhadas (seção `.data` e seção `bss`) pois conceitualmente elas residem em endereços “fixos” e devem ser alocadas uma em cada processo.

As variáveis globais de um objeto compartilhado podem ser de escopo local (ou seja, declaradas no objeto compartilhado e utilizadas unicamente pelo objeto compartilhado) ou podem ser de escopo global (ou seja, declaradas no objeto compartilhado e acessadas pelo processo). Este livro utilizará o termo variável global interna para indicar o primeiro tipo de variável (de escopo local) e variável global externa para indicar o segundo tipo de variável (escopo global).

Este problema torna-se mais complicado ao lembrar que as variáveis contidas em um processo não podem ser alteradas por outro processo, ou seja, elas não podem ser colocadas no espaço de endereçamento de outro processo.

Como pode ser visto pelo que foi exposto acima, é possível dividir os problemas logísticos envolvendo bibliotecas compartilhadas em dois grupos: execução de desvios e acesso às variáveis globais.

Várias soluções foram propostas para resolver estes problemas, e muitas delas estão descritas em [25], que faz uma análise bastante ampla.

Como o escopo deste livro é mais restrito do que o escopo daquele, o que será apresentado a seguir concentra-se em descrever as soluções adotadas no modelo ELF utilizado no Linux e com processador AMD64.

10.2 Ferramentas do *Hardware*

As instruções assembly que foram apresentadas ao longo deste livro apresentam alguns problemas para funcionarem em bibliotecas compartilhadas. Felizmente existe um conjunto de instruções apropriadas em quase todos os processadores modernos, e esta seção apresenta algumas presentes no AMD64.

Este conjunto será apresentado em dois grupos: instruções de desvio e instruções de acesso a dados, abordados nas Seções 10.2.1 e 10.2.2 respectivamente.

10.2.1 Chamadas de procedimento e instruções de desvio

Relembre do início deste capítulo que uma biblioteca compartilhada pode ser carregada em endereços virtuais diferentes. Relembre também que as instruções de desvio utilizadas até o momento neste livro (`jmp <endereço>` e `call <endereço>`) utilizam como parâmetro um endereço fixo e que por isto, elas não são apropriadas para uso em bibliotecas compartilhadas.

Esta seção descreve duas instruções assembly especiais e como elas são utilizadas para contornar esta situação. A primeira instrução especial, apresentada na seção 10.2.1.1, permite que o desvio seja relativo ao endereço da instrução corrente, denominado genericamente de desvio relativo ao PC (que no AMD64 é endereçamento relativo ao `%rip`). A segunda instrução especial, apresentada na seção 10.2.1.2 e denominada desvio indireto, permite que o alvo do desvio seja indicado pelo conteúdo de uma posição de memória.

10.2.1.1 Desvio Relativo ao PC

A instrução de desvio relativo ao PC está presente praticamente todos os processadores de propósito geral modernos. Considere um arquivo executável, mais especificamente a seção `.text` deste arquivo executável. O código executável do programa começa no primeiro endereço da seção `.text` e o endereço de cada uma das instruções subsequentes pode ser calculado considerando o início da instrução atual acrescido do tamanho da instrução atual.

Seja i o endereço de uma instrução. Conforme o que foi explicado acima, o endereço da próxima instrução pode ser determinado somando o endereço da instrução atual com o tamanho da instrução atual. De maneira análoga o endereço da instrução anterior pode ser calculado subtraindo o endereço da instrução atual do tamanho da instrução anterior. Este processo permite determinar o endereço de qualquer instrução antes e depois de uma instrução qualquer.

As coisas ficam mais interessantes quando se lembra que o registrador `%rip` sempre aponta para a o endereço da próxima instrução. Suponha uma instrução de desvio cujo endereço alvo está localizado 116 bytes acima do endereço da próxima instrução. Este endereço pode facilmente ser calculado com a fórmula $\%rip + (-116)_{10}$, ou $\%rip + (cf)_{16}$ (observe que $(cf)_{16}$ está em complemento de dois).

Agora podemos especificar a instrução de desvio relativo ao PC: ela segue o modelo `jmp <deslocamento>`⁵, que soma o parâmetro ao valor corrente de `%rip`, ou seja:

⁵ miseravelmente o mnemônico da instrução de desvio relativo ao PC é o mesmo do mnemônico do desvio absoluto. Eles são diferenciados pelos códigos de operação.

$$\%rip \leftarrow \%rip + <deslocamento>$$

As instruções de desvio relativo ao PC são frequentemente utilizadas na tradução de estruturas de linguagem de programação, como repetição (`while`, `repeat`, `for`, etc.) e condicionais (`if`, `case`, etc.).

Vamos agora examinar isto na prática. Para tal, examine o comando `while` nas linhas 13, 14 e 15 do algoritmo 10.3) (b.c) (página 152) copiado abaixo:

```
...
13:  while ( gB_ext < gB_int ) {
14:      gB_ext = gB_ext +1;
15:  }
...
```

O código assembly gerado pelo compilador para este comando `while` pode ser visto ao examinar o código objeto:

```
> objdump -d b.o
...
74:      48 8b 05 00 00 00 00      mov     0x0(%rip),%rax        # 7b <b+0x7b>
7b:      48 8b 00                  mov     (%rax),%rax
7e:      48 8d 50 01              lea     0x1(%rax),%rdx
82:      48 8b 05 00 00 00 00      mov     0x0(%rip),%rax        # 89 <b+0x89>
89:      48 89 10                  mov     %rdx,%rax
8c:      48 8b 05 00 00 00 00      mov     0x0(%rip),%rax        # 93 <b+0x93>
93:      48 8b 10                  mov     (%rax),%rdx
96:      48 8b 05 00 00 00 00      mov     0x0(%rip),%rax        # 9d <b+0x9d>
9d:      48 8b 00                  mov     (%rax),%rax
a0:      48 39 c2                  cmp     %rax,%rdx
a3:      7c cf                    jl      74 <b+0x74>
```

O endereço `0x74` é o endereço de início do laço, enquanto que o endereço `0xa3` é o último endereço do laço, o local em que ocorre o desvio incondicional para o início do laço.

A instrução contida no endereço `0xa3` é um desvio (*jump if less*) com código de operação `0xc2` e parâmetro `0xcf`. Fica claro que o parâmetro não é um endereço absoluto do início do laço. O volume 3 do manual do AMD [18] diz que este código de operação corresponde a uma instrução do tipo *jump near*, ou seja desvio para alvo próximo.

Agora vamos examinar como esta instrução funciona em tempo de execução.

Suponha que a comparação efetuada em `0xa0` ative o bit *less*. Ao executar a instrução no endereço `0xa3`, o `%rip` irá conter o valor `0xa5`, e como o desvio deve ser executado, o endereço da próxima instrução será obtido pela fórmula abaixo:

$$\%rip \leftarrow \%rip + <deslocamento>$$

$$\%rip \leftarrow 0x00a5 + 0xcf \text{ (Em complemento de dois)}$$

$$\%rip \leftarrow 0x004f$$

O que retorna o fluxo de execução ao início do laço.

As instruções de desvio relativo ao PC podem substituir as instruções de desvio com o endereçamento absoluto contanto que o endereço alvo esteja dentro do intervalo especificado pelo parâmetro. No exemplo apresentado, o parâmetro é um inteiro de 8 bits, e só pode ser utilizado se o endereço alvo do desvio estiver no intervalo $[-2^8 + \%rip, \%rip + 2^8 - 1]$, ou seja, no intervalo $[-128, 127]$ bytes distante do `%rip`. Existem variações desta instrução onde o parâmetro é um inteiro de 16 bits ou de 32 bits, permitindo que o endereço alvo esteja até 2G bytes distante da instrução atual.

10.2.1.2 Desvio Indireto

As instruções de desvio relativo ao PC resolvem os problemas de desvio das construções típicas de linguagens de programação utilizados dentro da própria biblioteca. Porém, elas não podem ser utilizadas quando o endereço alvo está muito distante, como por exemplo quando está sendo executado o código no segmento de processo e o alvo é uma biblioteca compartilhada (veja figura 28).

Para este caso é utilizado outro tipo de instruções de desvio, denominadas instruções de desvio indireto. Nestas instruções, o parâmetro é um endereço cujo conteúdo é o alvo do desvio.

Para diferenciar a instrução de desvio indireto da instrução de desvio convencional, a primeira é representada com um asterisco à frente do parâmetro: `jmp *<endereço>`.

Apesar de semelhantes na forma, as instruções de desvio absoluto e desvio indireto tem funcionalidades diferentes conforme indicado na tabela 18. Enquanto a instrução de desvio absoluto coloca em `%rip` o conteúdo do parâmetro (neste caso indicado com `<endereço>`), a instrução de desvio indireto coloca em `%rip` o conteúdo da posição de memória indicado no parâmetro.

<code>jmp <endereço></code>	<code>%rip ← <endereço></code>
<code>jmp *<endereço></code>	<code>%rip ← M[<endereço>]</code>

Tabela 18 – Funcionalidade das instruções de desvio absoluto e desvio indireto

Agora vamos analisar como utilizar as instruções de desvio indireto. Considere o algoritmo 10.1 como exemplo. Ele contém um código extremamente complicado para fazer uma coisa simples: imprimir “-2”. Porém, estamos aqui pela viagem e não pelo destino, e neste caso podemos dizer que este código é realmente uma viagem pois são necessárias várias explicações antes de prosseguir.

As primeiras linhas de programa contém quatro procedimentos denominados `soma` (linha 3), `subt` (linha 7), `mult` (linha 11) e `divi` (linha 15). Cada um deles executa a operação aritmética indicada sobre os parâmetros passados por valor.

A linha 22 declara a variável `vetPtrFunc`, um vetor de 4 posições tal que cada posição contém um endereço (ponteiro) para um procedimento.

As linhas 27 até 30 copiam os endereços dos procedimentos das linhas 3, 7, 11 e 15 para `vetPtrFunc`.

Após a execução da linha 30, teremos algo semelhante à figura 29.

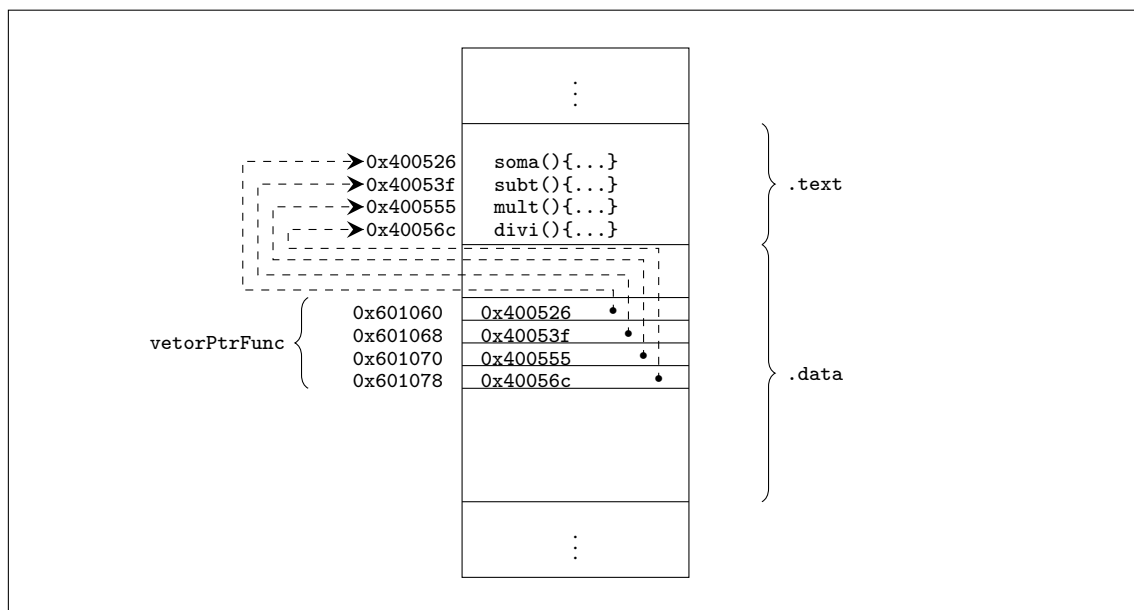


Figura 29 – Execução do algoritmo 10.1

Os endereços indicados na figura foram obtidos via análise da execução no `gdb`. O endereço de `vetPtrFunc[0]` é `0x601060` e os demais elementos do vetor estão distanciados de 8 em 8 bytes (ou seja, cada posição do vetor responde endereço de 64 bits) na seção `.data`. Os procedimentos `soma`, `subt`, `mult` e `divi` foram mapeados na seção `.text`, mais especificamente nos endereços `0x400526`, `0x40053f`, `0x400555` e `0x40056c`.

Observe que cada posição de `vetPtrFunc` contém o endereço dos respectivos procedimentos. Posto em outras palavras, cada posição de `vetPtrFunc` aponta para um procedimento.

Agora, vamos explorar o conceito de vetor de ponteiros para procedimento e analisar como ele pode ser usado em bibliotecas compartilhadas. Uma ideia é criar um vetor semelhante ao apresentado na figura 29, onde cada posição

```

1  #include <stdio.h>
2
3  long int soma(long int a, long int b){
4      return a + b;
5  }
6
7  long int subtr(long int a, long int b){
8      return a - b;
9  }
10
11 long int mult(long int a, long int b){
12     return a * b;
13 }
14
15 long int divi(long int a, long int b){
16     if(b)
17         return a / b;
18     else
19         return 0;
20 }
21
22 long int (*vetPtrFunc[4]) (long int x, long int y);
23
24 long int main(long int argc, char **argv){
25     long int i, j;
26
27     vetPtrFunc[0] = soma;
28     vetPtrFunc[1] = subtr;
29     vetPtrFunc[2] = mult;
30     vetPtrFunc[3] = divi;
31
32     // operacao: i=(1+2) - (4/3)*5
33
34     i = vetPtrFunc[0](1,2);
35     j = vetPtrFunc[3](4,3);
36     j = vetPtrFunc[2](j,5);
37     i = vetPtrFunc[1](i,j);
38     printf ("%ld\n", i);
39 }

```

Algoritmo 10.1 – Programa que acessa as funções a partir de um vetor de ponteiros

aponta para um procedimento na biblioteca compartilhada. Uma vez estabelecidos os respectivos endereços no vetor, basta utilizar uma instrução de desvio indireto onde o parâmetro é o índice do vetor, algo como `call *vetPtrFunc[i]`. Observe que o número de entradas é calculado pelo ligador estático (veja exercício 10.2).

O problema é preencher o vetor com os endereços dos procedimentos da biblioteca compartilhada. Uma possibilidade é o carregador preencher todo o vetor antes de iniciar a execução de programa, mas neste caso o custo computacional só será recompensado se todos os procedimentos forem chamados. Se o programa utilizar centenas de procedimentos compartilhados, o programa poderá gastar alguns segundos só para preencher a tabela, mas se o programa finalizar execução sem chamar nenhum dos procedimentos, este tempo terá sido em vão.

A situação ideal é aquela onde somente os procedimentos efetivamente chamados são colocados no vetor, e isto é possível com o seguinte algoritmo: a primeira vez que o procedimento for chamado, verifica-se se o endereço do procedimento já foi colocado no vetor. Caso já tenha sido, faz o desvio indireto usando aquela entrada. Caso o endereço não tenha sido colocado no vetor, então é necessário: (1) encontrar o endereço do procedimento na biblioteca compartilhada (2) colocar este endereço no vetor e (3) fazer o desvio indireto utilizando a recém inserida entrada na tabela.

O ato de descobrir o endereço de um procedimento (ou variável global) é chamado *binding*⁶ e existe um procedimento chamado `resolver()` no ligador dinâmico `ld-linux-x86-64.so.2` projetado especificamente para fazer o *binding*.

Com esta informação, podemos reescrever o algoritmo da seguinte forma:

⁶ neste contexto, a melhor tradução é “vínculo”, mas adotaremos o termo em inglês

1. quando o procedimento for chamado, verificar se já foi feito o *binding*;
2. se já foi, faz o desvio indireto usando aquela entrada;
3. se não foi, chama o `resolver()` no ligador dinâmico `ld-linux-x86-64.so.2` que fará o *binding* naquela entrada. Por fim faz o desvio indireto usando aquela entrada;

O algoritmo acima envolve um comando condicional, que pode ser substituído por uma versão mais rápida. Nesta versão, todas as entradas do vetor são inicialmente preenchidas com o endereço do `resolver()`, ou seja: sempre tem um procedimento é chamado pela primeira vez, o fluxo é desviado para o `resolver()`. Quando entra em execução, o `resolver()` utiliza dois parâmetros, o nome do procedimento a ser procurado, e o índice do vetor o que deve receber o endereço de procedimento.

Por exemplo, suponha que o procedimento `proc1()` tenha sido associado à *i*-ésima entrada do vetor. Considerando o exemplo acima, `vetPtrFunc[i]` é a entrada que deveria conter o endereço de `proc1()` mas como ele ainda não foi executado nenhuma vez, contém o endereço do `resolver()`.

Na primeira chamada de `proc1()`, quem será executado é o `resolver()`, que irá procurar pelo procedimento correspondente e irá escrever o endereço de `proc1()` em `vetPtrFunc[i]`. Após fazer tudo isto, o fluxo será desviado para a entrada recém incluída.

Na segunda vez em que o procedimento `proc1()` for chamado, ele irá desviar o fluxo de execução direto para o endereço de `proc1()`. Observe que essa solução minimiza o número de chamadas ao `resolver()` já que só os procedimentos efetivamente chamados irão executá-lo, potencialmente melhorando o desempenho do programa.

Este mecanismo de atrasar o *binding* até o momento em que for absolutamente necessário é chamado de *lazy binding*. Aliás, este não é o único caso em computação. *Lazy*⁷ é um termo genérico para descrever uma família de otimizações onde o trabalho a ser feito é atrasado até o momento onde ele é efetivamente necessário. Exemplos de mecanismos “preguiçosos” incluem *copy-on-write*[11] (utilizado principalmente em sistemas operacionais) e *lazy evaluation*[24] (utilizado em linguagens de programação).

Agora vamos interromper a discussão sobre as instruções de desvio para abordar as instruções de acesso a dados. Na analogia com degraus, descemos um degrau na escadaria que explica como trabalhar com desvios em bibliotecas compartilhadas e ao invés de continuar a descida nesta escadaria, vamos descer também um degrau na escadaria que explica como trabalhar com variáveis globais em bibliotecas compartilhadas.

O próximo degrau da escadaria que explica como trabalhar com desvios em bibliotecas compartilhadas aborda detalhes de implementação no linux/AMD64, e está na seção 10.3.2.

10.2.2 Instruções de acesso a dados

A seção 10.2.1 mostrou que as instruções assembly utilizadas para implementar desvio de fluxo de execução tem um problema grave: o endereço alvo, indicado no parâmetro da instrução (`jmp <endereço>` e `call <endereço>`) é um endereço fixo. Como as bibliotecas compartilhadas podem ser carregadas em qualquer endereço virtual, estas instruções com endereçamento fixo não poderiam ser utilizadas em bibliotecas compartilhadas. Naquela seção foram apresentadas duas variações das instruções de desvio que são apropriadas para utilização em bibliotecas compartilhadas: instruções com desvio relativo ao PC e instruções de desvio indireto.

As instruções de acesso às variáveis globais sofrem do mesmo mal, pois utilizam o modo de endereçamento chamado “endereçamento direto”, e conforme apresentado na seção B.4 neste modo de endereçamento, o endereço alvo é fixo (tanto que é incluído na própria instrução). Por exemplo na instrução `movq $1, A` a constante 1 é armazenada no endereço de memória reservado à variável A. Supondo que o endereço da variável A é `0x600000`, o endereço alvo poderia ser explicitamente representado na instrução, ou seja: `movq $1, 0x600000`.

Porém, diferente do caso dos desvios, nenhuma nova instrução precisou ser acrescentada para o acesso a variáveis globais de bibliotecas compartilhadas.

O mecanismo utilizado para acessar as variáveis globais de uma biblioteca compartilhada usa o mesmo princípio de acessar variáveis locais e parâmetros na pilha visto no capítulo 3. A diferença é que enquanto lá o endereçamento era feito relativo ao registrador `%rbp`, aqui o registrador utilizado é o `%rip`. Chamaremos este modo de endereçamento de “endereçamento indireto via `%rip`”.

Vamos agora analisar como este modo de endereçamento pode ser utilizado para implementar o acesso a variáveis globais de bibliotecas compartilhadas.

Relembre da seção 10.2.1 que o código executável tem tamanho fixo e que é possível conhecer o endereço de qualquer instrução a partir do endereço da instrução atual somando (ou subtraindo) o tamanho da instrução corrente (ou anterior).

⁷ “preguiçoso”

Como a seção `.data` começa em um endereço conhecido (considere como referência que começa no endereço `0x600000`), também é possível calcular a quantidade de bytes que separa a instrução atual da primeira variável global armazenada na seção `.data`, e isto pode ser calculado pelo ligador estático quando ele está gerando o arquivo objeto.

Assim, se a instrução apontada por `%rip` estiver no endereço `0x400000` e a primeira variável global da seção `.data` for mapeada no endereço `0x600000`, então a variável global pode ser acessada de duas formas:

1. `movq $1, 0x600000`
2. `movq $1, 0x200000(%rip)`

A primeira instrução foi utilizada extensivamente ao longo do livro, e não é apropriada em bibliotecas compartilhadas. A segunda utiliza o modo de endereçamento indireto via `%rip`. O conteúdo de `%rip` (`0x400000`) é somado à constante `0x200000`, e o mesmo endereço alvo é obtido (`0x600000`).

Isto não é muito fácil de entender de primeira, por isto vamos ilustrar esta ideia num exemplo onde:

- a seção `.text` tem `0x0100` bytes e foi carregada para o endereço `0x0400`;
- a variável global `varGlobal1` é a primeira da seção `.data` e foi carregada no endereço `0x0500`;

Suponha que a primeira instrução da seção `.text`, é uma instrução que copia a constante 44 para `varGlobal1`.

Antes de explicar como uma biblioteca compartilhada usa esta instrução, vamos considerar primeiro o que ocorreria na geração da versão estática. Neste caso, o ligador sabe, já em tempo de ligação, que em tempo de execução a `varGlobal1` será mapeada no endereço `0x0500`. Logo, o ligador estático poderia utilizar qualquer uma das seguintes instruções:

```
movq $44, varGlobal1    ou
movq $44, 0x0500
```

Porém no caso de bibliotecas compartilhadas, o endereço de `varGlobal1` é desconhecido quando o código objeto é gerado (durante a execução do ligador estático). A única informação disponível para o ligador estático é a quantidade de bytes que irão separar a seção `.text` da seção `.data` em tempo de execução. No exemplo acima a quantidade de bytes que irão separar a seção `.text` da seção `.data` é de `0x0100` bytes.

Isto significa também que é possível calcular distância entre qualquer instrução da seção `.text` e a primeira variável global na seção `.data`. Aliás, como o tamanho das variáveis globais também é conhecido, a frase acima pode ser generalizada para: “é possível calcular distância entre qualquer instrução da seção `.text` e qualquer variável global na seção `.data`”.

Considere que a instrução citada é a primeira na seção `.text`, e suponha que ela ocupe 4 bytes. Neste cenário, quando ela for executada, o valor de `%rip` estará distante `0x96` bytes do endereço de `varGlobal1` (lembre-se que `%rip` já aponta para a próxima instrução), o que sugere a instrução a seguir:

```
movq $44, $0x96(%rip)
```

Observe que se a próxima instrução também acessar `varGlobal1`, então `%rip` será diferente, e por isso o valor será menor do que `0x96`. Como regra geral, considere que a instrução será sempre algo como

```
movq <origem>, $(desloc)(%rip)
```

Antes de finalizar esta seção, é importante mencionar alguns detalhes:

1. o ligador estático procura colocar a seção `.data` imediatamente após a seção `.text`. Isto permite que as distâncias entre a seção `.text` e seção `.data` sejam conhecidas pelo ligador dinâmico, que precisa somente fazer a relocação nas instruções que utilizam o endereçamento indireto relativo ao `%rip`.
2. “colocar a seção `.data` imediatamente após a seção `.text`” não implica dizer que a seção `.data` começa no byte seguinte ao último byte da seção `.text`. Lembre-se que as duas seções devem ser mapeadas em páginas virtuais com propriedades de leitura escrita e execução diferentes. Assim, em tempo de compilação, o ligador pode deixar indicado no arquivo objeto que a seção `.data` deve ser mapeado em outra página virtual (não obrigatoriamente a página virtual imediatamente seguinte), podendo deixar um espaço (por vezes grande) entre as duas seções.

Neste ponto atingimos, na escadaria de variáveis globais, o mesmo degrau da escadaria de chamadas de desvios indicado ao final da seção 10.2.1.

O próximo passo é descer um degrau em cada escadaria, o que é feito na seção 10.3. Assim, interrompemos a discussão de como acessar as variáveis globais de bibliotecas compartilhadas para tratar de detalhes de implementação no ELF/Linux/AMD64. O tema será retomado na seção 10.3.3 que descreve como o ELF/AMD64/linux utiliza as instruções e conceitos apresentados aqui para acessar variáveis globais.

10.3 Implementação no ELF/Linux/AMD64

Vamos agora descer mais um degrau e descrever o que ocorre em tempo de execução quando um programa que usa uma biblioteca compartilhada é colocado em execução no ELF do linux, especificamente no processador AMD64.

Para tal, a seção 10.3.1 apresenta um programa que será utilizado na seção 10.3.2 (que desce um degrau na escadaria de procedimentos) e na seção 10.3.3 (que desce um degrau na escadaria de variáveis globais).

10.3.1 Exemplo de trabalho

Esta seção apresenta três programas na linguagem C: a.c (algoritmo 10.2), b.c (algoritmo 10.3) e main.c (algoritmo 10.4).

```

1 #include <stdio.h>
2
3 long int gA_ext;
4 long int gA_int;
5
6 void funcao_interna_em_a (long int x) {
7     gA_int = gA_int + x;
8 }
9
10 void a (char* s) {
11     gA_ext=16;
12     printf("gA_ext: (Valor=%02ld) (Endereco=%p)\n", gA_ext, &gA_ext);
13
14     gA_int=15;
15     printf("gA_int: (Valor=%02ld) (Endereco=%p)\n", gA_int, &gA_int);
16
17     funcao_interna_em_a(15);
18     printf("funcao_interna_em_a: (Endereco=%p)\n", &funcao_interna_em_a);
19 }
```

Algoritmo 10.2 – Arquivo a.c

```

1 #include <stdio.h>
2
3 long int gB_ext;
4 long int gB_int;
5
6 void b (char* s) {
7     gB_ext=14;
8     printf("gB_ext: (Valor=%02ld) (Endereco=%p)\n" , gB_ext, &gB_ext);
9
10     gB_int=13;
11     printf("gB_int: (Valor=%02ld) (Endereco=%p)\n" , gB_int, &gB_int);
12
13     while (gB_ext < gB_int) {
14         gB_ext = gB_ext +1;
15     }
16 }
```

Algoritmo 10.3 – Arquivo b.c

Gera-se o arquivo executável compartilhado com a sequência abaixo:

```

> gcc -fPIC -c -o a.o a.c
> gcc -fPIC -c -o b.o b.c
> gcc -fPIC -shared a.o b.o -o libMySharedLib.so
> gcc main.c -L. -lMySharedLib -o main
```

Enquanto o arquivo executável estático poderia ser gerado com a sequência abaixo:


```

1  #include <stdio.h>
2
3  extern long int gA_ext;
4  extern long int gB_ext;
5
6  long int gMain;
7
8  void a (char* s);
9  void b (char* s);
10
11 long int main ()
12 {
13     gA_ext=15;
14     gB_ext=14;
15
16     printf ("Procedimento main: (Endereco=%p)\n", main);
17     printf ("Procedimento a: (Endereco=%p)\n", a);
18     printf ("Procedimento b: (Endereco=%p)\n", b);
19     printf ("Procedimento printf: (Endereco=%p)\n", printf);
20
21     a("dentro de a\n");
22     b("dentro de b\n");
23 }

```

Algoritmo 10.4 – Arquivo main.c

```

> gcc -c -o a-static.o a.c
> gcc -c -o b-static.o b.c
> ar -src libStatic.a ?-static.o
> gcc -static main.c -o main-static -lStatic -L.

```

As próximas seções irão explicar o funcionamento do carregador dinâmico. Para tal serão utilizados, como referência, os arquivos objeto `a.o` e `b.o` e o executável `main` gerados com os comandos acima com ênfase na localização e uso de variáveis globais e chamadas de procedimento conforme destacado a seguir:

variáveis globais: o programa contém quatro variáveis globais divididas em dois grupos:

globais internas: As variáveis `gA_int` e `gB_int` são globais visíveis em `a.c` e `b.c` respectivamente, e só são visíveis naqueles arquivos e não em outros, em especial não são visíveis em `main.c`.

globais externas: As variáveis `gA_ext` e `gB_ext` foram declaradas em `main.c` como `extern` (linhas 3 e 4 respectivamente). Isto significa que elas são visíveis em `main.c`, porém não é ele quem aloca espaço para elas e sim outro arquivo (no caso de `gA_ext`, está alocado na linha 3 de `a.c`).

chamadas de procedimento: Os arquivos apresentam chamadas de procedimento que podem ser divididas em dois grupos:

programa principal para as bibliotecas: é o caso das linhas 21 e 22 do programa `main` (algoritmo 10.4) que chama os procedimentos `a()` e `b()` respectivamente.

chamadas internas às bibliotecas: é o caso da chamada ao procedimento `funcao_interna_em_a()` declarada na linha 6 e chamada da linha 17 de `a.c` (algoritmo 10.4). Esta função não é conhecida em `main.c` e nem em `b.c` (algoritmos 10.4 e 10.3 respectivamente).

Após gerar o executável é possível visualizar as bibliotecas compartilhadas usadas pelo programa `main` com o comando `ldd`.

```

> ldd main
linux-vdso.so.1 => (0x00007fff047f3000)
libMySharedLib.so => ./libMySharedLib.so (0x00007f907f6a9000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f907f2b9000)
/lib64/ld-linux-x86-64.so.2 (0x0000557d72089000)

```

linux-vdso.so.1 É uma biblioteca que disponibiliza algumas chamadas ao sistema na área do usuário para melhorar o desempenho⁸.

libMySharedLib.so A biblioteca compartilhada que contém os objetos a.o e b.o.

libc.so.6 Biblioteca contendo as implementações de dezenas de procedimentos comuns em C tais como `printf()` e `scanf()`.

/lib64/ld-linux-x86-64.so.2 É o responsável por mapear os endereços das demais bibliotecas compartilhadas e dinâmicas. É lá que está o `resolver()`.

Ao carregar o programa `main`, o carregador verifica quais as quatro bibliotecas compartilhadas que precisam ser carregadas e solicita que o ligador dinâmico as carregue. Este, por sua vez, verifica se elas já estão disponíveis em memória (carregadas por algum outro programa). Se estiverem, utiliza as páginas já disponíveis e se não estiverem, aloca a biblioteca.

10.3.2 Procedimentos

A seção 10.2.1 apresentou instruções e modos de endereçamento que permitem a implementação de desvios bibliotecas compartilhadas. As instruções de desvio relativo ao PC (seção 10.2.1.1) são utilizadas na tradução de construções típicas de linguagens de programação como `while` e `if`, ou para desvios para alvos próximos. Já as instruções de desvio indireto apresentadas na seção 10.2.1.2 são utilizadas quando o alvo está mais distante.

O problema que ocorre quando uma biblioteca precisa chamar um procedimento local à biblioteca é resolvido facilmente utilizando as instruções de desvio relativo ao PC.

Já o problema que ocorre quando o programa principal chama uma biblioteca é resolvido com o auxílio das instruções de desvio indireto. Porém neste caso, a implementação não é trivial. O restante desta seção explica como isto é feito.

Relembre que a figura 29 sugeriu a criação de um vetor de apontadores para procedimentos. A ideia era que cada entrada do vetor aponte para um procedimento da biblioteca compartilhada. Assim, a instrução de desvio do tipo `call proc()` seria substituída por algo como `call *vetPtrFunc[i]`, onde *i* corresponde ao índice do vetor que contém o endereço do procedimento em questão.

Tanto os arquivos executáveis quanto os arquivos objeto podem conter um vetor de apontadores como o `vetPtrFunc`. Para tal, existe uma seção específica do modelo ELF chamada `.got.plt`⁹ onde as entradas do vetor de apontadores `vetPtrFunc` (`vetPtrFunc[0]`, `vetPtrFunc[1]`, etc.) são representadas no arquivo objeto como `.got.plt[0]`, `.got.plt[1]`, etc.

Quando o carregador inicia a execução de um processo, ele coloca as bibliotecas compartilhadas na memória (se já não estiverem lá, claro). Ele também aloca espaço para o vetor de ponteiros de acordo com a quantidade especificada de posições que ele encontrar na seção `.got.plt`. Por fim, ele preenche a primeira posição do vetor com o endereço de uma estrutura de dados com informações sobre a localização das bibliotecas compartilhadas utilizadas por este processo (lembre-se que o `resolver()` é executado por qualquer processo, e precisa das informações de contexto saber quem é quem), a segunda posição com o endereço do `resolver()` e preenche os demais de acordo com a regra que será apresentada um pouco mais à frente.

Vamos interromper esta explicação para apresentar outro aspecto incluído no arquivo ELF, intimamente relacionado com o vetor de apontadores para procedimentos: a seção `.plt`.

Esta seção contém instruções assembly, mas especificamente rótulos associados aos procedimentos das bibliotecas compartilhadas. Para cada procedimento da biblioteca compartilhada que o programa chamar, haverá uma entrada com rótulo semelhante ao nome do procedimento na seção `.plt`. Por exemplo, as entradas para os procedimentos `b()` e `printf()` estão indicadas abaixo.

```
> objdump -S main
...
Disassembly of section .plt:

000000000400620 <b@plt-0x10>:
400620:    ff 35 e2 09 20 00    pushq 0x2009e2(%rip)    # 601008 <_GLOBAL_OFFSET_TABLE_+0x8>
400626:    ff 25 e4 09 20 00    jmpq  *0x2009e4(%rip)    # 601010 <_GLOBAL_OFFSET_TABLE_+0x10>
40062c:    0f 1f 40 00          nopl  0x0(%rax)
```

⁸ mais detalhes em <http://man7.org/linux/man-pages/man7/vdso.7.html>

⁹ GOT=Global Offset Table e PLT=Procedure Linkage Table

```

000000000400630 <b@plt>:
  400630:    ff 25 e2 09 20 00    jmpq    *0x2009e2(%rip)    # 601018 <_GLOBAL_OFFSET_TABLE_+0x18>
  400636:    68 00 00 00 00       pushq   $0x0
  40063b:    e9 e0 ff ff ff       jmpq    400620 <_init+0x28>
000000000400640 <printf@plt>:
  400640:    ff 25 da 09 20 00    jmpq    *0x2009da(%rip)    # 601020 <_GLOBAL_OFFSET_TABLE_+0x20>
  400646:    68 01 00 00 00       pushq   $0x1
  40064b:    e9 d0 ff ff ff       jmpq    400620 <_init+0x28>
...
  4007ef:    e8 3c fe ff ff       callq   400630 <b@plt>
...

```

Ao invés de a chamada de um procedimento ser realizada com o comando `call *<endereço>` (onde <endereço> corresponde a uma entrada na `.got.plt`), a chamada é realizada com o comando `call <proced>@plt` cujo primeiro comando faz `jmp *<endereço>`.

Cada entrada na seção `.plt` é iniciada com um rótulo que segue o padrão `<proced>@plt`, onde <proced> corresponde ao nome do procedimento na biblioteca compartilhada. Assim, no nosso programa de trabalho, há entradas `a@plt`, `b@plt` e `printf@plt`.

O código apresentado contém muitos detalhes que tiram o foco do essencial. Por esta razão vamos primeiro analisar o algoritmo 10.5 que apresenta o modelo esquemático de uma entrada qualquer n ($n > 0$) da seção `.plt` antes de voltar ao código assembly.

```

1 <procn@plt>:
2   jmp *got.plt[n]
3   pushq n
4   jmp zero@plt

```

Algoritmo 10.5 – Procedimento genérico da seção `.plt`

O código é bem compacto, e a função de cada uma das quatro linhas é explicada a seguir:

linha 1 rótulo que indica a n -ésima entrada na seção `plt`. Este rótulo segue o padrão `<proced>@plt`. Qualquer chamada a um procedimento em biblioteca compartilhada que seria traduzida normalmente para `call <proced>` será traduzida para `call <proced>@plt`.

linha 2 desvia indireto para o alvo indicado em `got.plt[n]`.

linha 3 empilha o número n , que indica qual entrada na `got.plt[n]` deve ser atualizado (na verdade, empilha $n - 2$ já que as duas primeiras entradas da `.got.plt` não podem ser alteradas e por isto não são consideradas).

linha 4 desvia o fluxo para a 0-ésima entrada na seção `.plt`.

O código está dividido em duas partes. As linhas 1 e 2 são executadas em todas as chamadas ao procedimento. Já as linhas 3 e 4 são executadas uma única vez, na primeira chamada.

Vamos analisar agora o que ocorre na primeira vez que `<procn@plt>` é chamado. O objetivo desta primeira chamada é colocar em `got.plt[n]` o endereço do procedimento `proc()` que será utilizada a partir da segunda chamada.

Inicialmente, `got.plt[n]` contém o endereço da instrução da linha 3 (`pushq n`). É muito esquisito, mas é isso mesmo: na primeira chamada, `jmp *got.plt[n]` desvia indiretamente o fluxo para a instrução seguinte: `pushq n`. Esta instrução empilha n , indicando qual entrada da `got.plt` deve ser alterada para, em seguida, desviar o fluxo para `jmp zero@plt`, que é a primeira entrada da seção `plt`.

É esta primeira entrada (denominada genericamente `plt0` ou `zero@plt`) que chama o `resolver()` como representado esquematicamente no algoritmo 10.6.

Em linhas gerais, ao entrar em `zero@plt`, já há um parâmetro empilhado (pela linha 3 de `procn`, `pushq n`). Assim, a execução em `zero@plt` chega na linha 2 que empilha o endereço de uma estrutura de dados apontada por `.got.plt[0]` e que na linha 3 desvia o fluxo para o procedimento cujo endereço está contido em `.got.plt[1]`.

Este procedimento apontado por `.got.plt[1]` é o `resolver()`. O `resolver()` usa os dois parâmetros empilhados (n e `.got.plt[0]`) para ajustar o endereço-alvo em `.got.plt[n]` e, ao final, desviar o fluxo para o procedimento-alvo. A questão dos parâmetros é interessante pois há os parâmetros do `resolver()` e do procedimento (veja exercício 10.13).

```

1 zero@plt:
2     empilha got.plt[0]
3     jmp *got.plt[1]

```

Algoritmo 10.6 – Procedimento zero@plt da seção .plt

Quando o `resolver()` iniciar a execução, o seu registro de ativação será diferente do tradicional, uma vez que foi primeiro empilhado o endereço de retorno (empilhado pela chamada `call procn@plt`) e **DEPOIS** os parâmetros. Ao finalizar a execução, é ele quem desempilha os parâmetros e ao final executa a instrução `ret` para desviar o fluxo para

Observe como termo o *lazy binding* se aplica neste caso: somente na primeira chamada é que o endereço do procedimento-alvo será efetivamente atualizado. Se não for chamado nenhuma vez, então não há necessidade de atualizar a sua entrada na `got.plt`.

Agora, podemos voltar ao código assembly reimpresso a seguir e analisar o que ocorre na prática.

```

> objdump -S main
...
Disassembly of section .plt:
000000000400620 <b@plt-0x10>:
    400620:    ff 35 e2 09 20 00    pushq 0x2009e2(%rip)    # 601008 <_GLOBAL_OFFSET_TABLE_+0x8>
    400626:    ff 25 e4 09 20 00    jmpq  *0x2009e4(%rip)    # 601010 <_GLOBAL_OFFSET_TABLE_+0x10>
    40062c:    0f 1f 40 00          nopl  0x0(%rax)
000000000400630 <b@plt>:
    400630:    ff 25 e2 09 20 00    jmpq  *0x2009e2(%rip)    # 601018 <_GLOBAL_OFFSET_TABLE_+0x18>
    400636:    68 00 00 00 00 00    pushq $0x0
    40063b:    e9 e0 ff ff ff      jmpq  400620 <_init+0x28>
000000000400640 <printf@plt>:
    400640:    ff 25 da 09 20 00    jmpq  *0x2009da(%rip)    # 601020 <_GLOBAL_OFFSET_TABLE_+0x20>
    400646:    68 01 00 00 00 00    pushq $0x1
    40064b:    e9 d0 ff ff ff      jmpq  400620 <_init+0x28>
...
    4007ef:    e8 3c fe ff ff      callq 400630 <b@plt>
...

```

Observe que cada entrada na seção `.plt` contém exatamente 16 bytes e por isto, o endereço de cada entrada é uma potência de 16 (0x400630, 0x400640, 0x400650, etc.).

A impressão não mostra `zero@plt`, mas sim `b@plt-0x10`. Em todos os códigos que analisei, o `zero@plt` ocupa 16 (0x10) bytes é representado pelo nome da primeira entrada sucedida por `-0x10` (que significa menos 0x10 bytes). Aliás, a última instrução (`nopl 0x0(%rax)`) só está lá para que somar quatro bytes a `b@plt-0x10`. A instrução `nop` significa *No Operation*.

A impressão gerada pelo `objdump` contém comentários úteis. Veja que a primeira instrução do procedimento `b@plt` é um desvio indireto para a entrada na `.got.plt` que contém o endereço do procedimento `b()`. A instrução é `jmpq *0x2009e2(%rip)`, uma instrução que combina desvio para endereço relativo e desvio indireto. O endereço relativo ao PC é obtido pelo operação `%rip + 0x2009e2`. O comentário ao lado direito da instrução indica o endereço resultante da soma: 0x601018.

Uma vez explicado o formato, voltamos a analisar o funcionamento. A configuração descrita acima pode ser visualizada esquematicamente no lado esquerdo da figura 30. Esta configuração mostra o estado da memória antes da primeira chamada ao procedimento `b()`. Observe que as duas primeiras entradas (`.got.plt[0]` e `.got.plt[1]`) contém endereços contidos em `/lib64/ld-linux-x86-64.so.2`, enquanto que `.got.plt[2]` contém o endereço 0x400636.

Agora, vamos usar esta figura para detalhar o que ocorre quando o procedimento `b@plt()` é chamado:

- (4) `<b@plt>` ponto de entrada.
- (5) `jmp *got.plt[2]` desvia o fluxo para a instrução indicada em `.got.plt[2]`, que neste momento aponta para a instrução seguinte (`pushq $0x0`).
- (6) `pushq $0x0` empilha primeiro parâmetro do `resolver()`, que indica qual entrada da `.got.plt` deve ser atualizada. Como as entradas `.got.plt[0]` e `.got.plt[1]` são reservadas, a entrada a ser atualizada é a entrada `.got.plt[2]`;
- (7) `jmp <zero@plt>` desvia o fluxo;

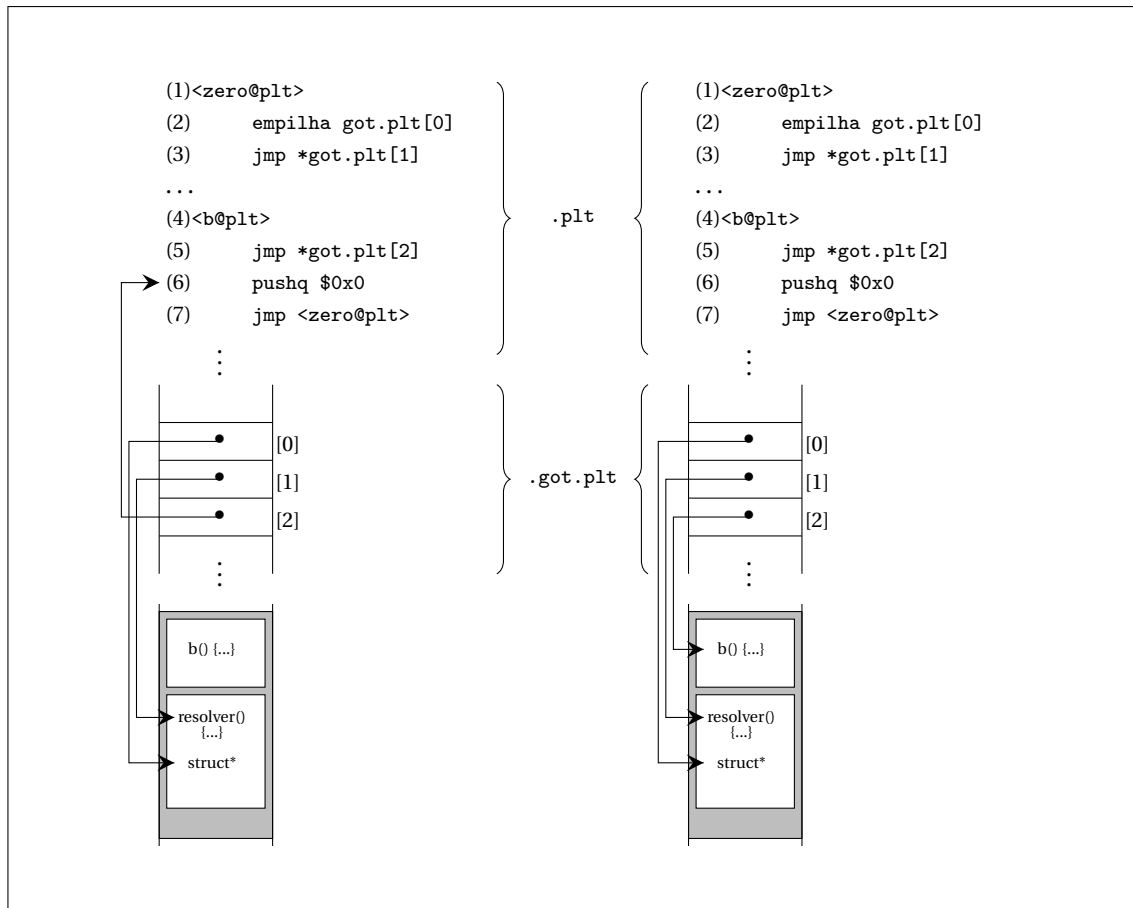


Figura 30 – Efeito da primeira chamada do procedimento “b()”, localizada em uma biblioteca compartilhada.

- (1) `<zero@plt>` alvo do desvio;
- (2) empilha `.got.plt[0]`, ou seja, empilha segundo parâmetro do `resolver()`.
- (3) `jmp *.got.plt[1]` desvia fluxo para o `resolver()`. A ação do `resolver()` muda a entrada em `.got.plt[2]` para o endereço de `b()` resultando no estado indicado no lado direito da figura 30. A última ação do `resolver()` é executar `jmp *.got.plt[2]`, ou seja, desviar o fluxo para `b()`.

Após a primeira chamada de `b@plt()`, as próximas não precisarão mais do `resolver()`. Considere uma nova chamada de `b@plt()`. Agora a configuração é aquela apresentada no lado direito da figura 30.

- (4) `<b@plt>` ponto de entrada.
- (5) `jmp *.got.plt[2]` desvia o fluxo para a instrução indicada em `.got.plt[2]`, ou seja, `b()`.

10.3.3 Variáveis Globais

Relembre que a seção 10.2.2 apresentou as instruções de endereçamento indireto via `%rip`. Estas instruções permitem atingir alvos distantes até 2 GBytes da instrução corrente, e é natural pensar que estas instruções são utilizadas para acessar as variáveis globais em bibliotecas compartilhadas.

Porém, se hoje 2 GBytes parece ser muito, em alguns anos pode não ser mais tanta coisa. Quando eu era um programador iniciante na década de 1980, 1 Mbyte de memória RAM era um latifúndio. Muitas das soluções projetadas naquela época foram otimistas considerando que aquele latifúndio não seria preenchido. Nem preciso dizer o que aconteceu.

Por esta razão, ao invés de utilizar as instruções que restringem a distância do alvo em 2 GBytes, uma solução mais genérica foi adotada. Esta solução pode vir a ser facilmente expandida para máquinas com memória “infinita”.

A ideia geral desta solução é utilizar uma região de memória contendo um vetor de ponteiros para as variáveis globais, denominada GOT (*Global Offset Table*). Cada entrada da GOT contém o endereço de uma variável global. Esta solução é muito parecida com aquela descrita anteriormente na seção 10.3.2, mas enquanto lá a seção `.got.plt` continha apontadores para procedimentos, aqui a seção `.got` contém apontadores para variáveis globais.

De forma análoga ao descrito anteriormente na seção 10.3.2, aqui também é utilizado o modo de endereçamento indireto via `%rip` para acessar uma entrada da `.got`.

O acesso a uma variável global é feito em dois passos. Como exemplo, considere a instrução que copia a constante 44 para a variável global `var_global`. O endereço desta variável está armazenada em `GOT[n]`.

```
movq $<desloc>(%rip), %rax # em $<desloc>(%rip) está a n-ésima entrada da GOT
movq $44, (%rax)          # armazena 44 na variável global.
```

Esta ideia está representada esquematicamente na figura 31, que mostra as três seções envolvidas no acesso a uma variável global em tempo de execução. A seção `.got` armazena os endereços das variáveis globais, neste caso de uma única variável global, localizada na seção `.data`.

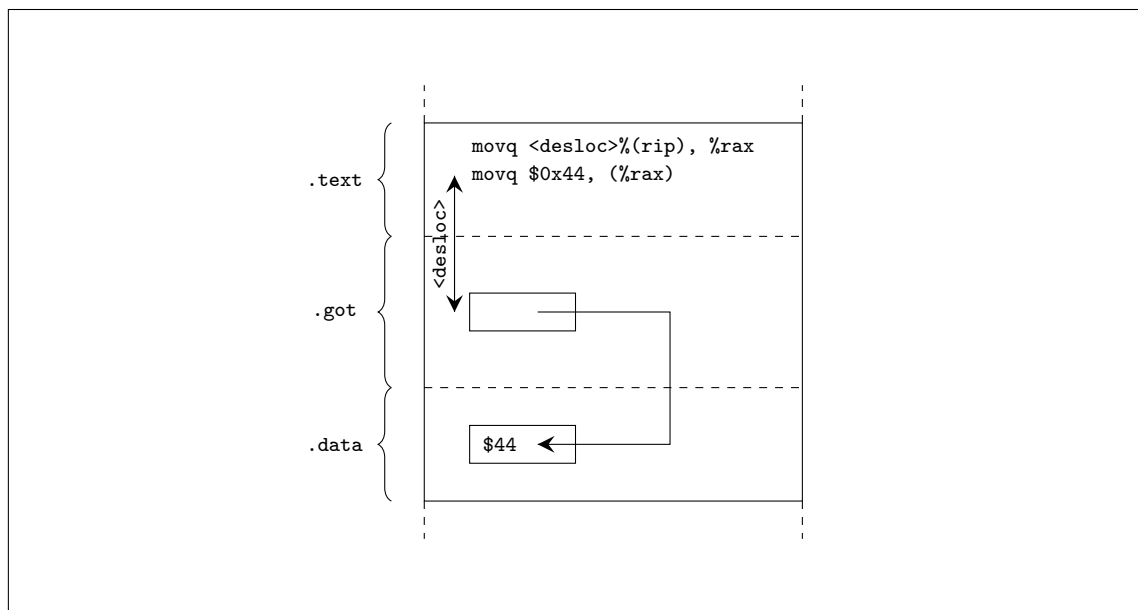


Figura 31 – Acesso às variáveis globais.

A seção `.text` contém as duas instruções citadas. a primeira instrução coloca o conteúdo da n-ésima posição de memória da GOT (distante `<desloc>` bytes de `%rip`) em `%rax`.

Após a execução desta primeira instrução, o registrador `%rax` irá conter o endereço de memória da variável global localizada na seção `.data`. Por fim, a instrução assembly seguinte armazena 44 na variável global.

Observe que a variável global reside na seção `.data`, enquanto que a GOT só contém o endereço da (ou ponteiro para a) variável global. É importante destacar que com este mecanismo, qualquer endereço da memória virtual pode ser acessado já que cada elemento da `got` tem tamanho de 64 bits.

Também é interessante destacar o trabalho que deve ser realizado pelo ligador estático, pelo carregador e pelo ligador em tempo de execução.

Ligador estático Ao gerar tanto o arquivo objeto quanto o arquivo executável, o ligador estático deve deixar indicada a presença de variáveis globais e que estas serão acessadas através da GOT. O código assembly gerado segue o modelo indicado acima para acessar as variáveis globais através da GOT. Porém o ligador estático não indica em que endereço virtual devem ser mapeadas as variáveis globais, e nem a seção `.got`.

Carregador o carregador dispara o ligador dinâmico, que carrega as bibliotecas dinâmicas, que:

1. aloca espaço para a seção `.got` e `.data` em endereços virtuais aleatórios.
2. faz o *fixup* da seção `.text`, ou seja ajust o `<desloc>` das instruções que acessam a memória usando endereçamento indireto via `%rip` de tal forma que a soma (`<desloc> + %rip`) aponte para a seção `.got`.

3. preenche os elementos da seção `.got` com os endereços das variáveis globais.

O mecanismo apresentado acima é flexível a ponto das variáveis globais serem mapeados em endereço virtuais diferentes em cada execução. Aliás, é exatamente isto que ocorre na execução do exemplo de trabalho da seção 10.3.1.

```
> ./main
...
gA_ext: (Valor=16) (Endereco=0x55aeb2980018)
gA_int: (Valor=15) (Endereco=0x7f5f11331030)
...
> ./main
...
gA_ext: (Valor=16) (Endereco=0x557fe3e33018)
gA_int: (Valor=15) (Endereco=0x7f6ff5b59030)
...
> ./main
...
gA_ext: (Valor=16) (Endereco=0x562282706018)
gA_int: (Valor=15) (Endereco=0x7f9f33f89030)
...
> ./main
...
gA_ext: (Valor=16) (Endereco=0x5578e809f018)
gA_int: (Valor=15) (Endereco=0x7f659c019030)
...
> ./main
...
gA_ext: (Valor=16) (Endereco=0x5617c9028018)
gA_int: (Valor=15) (Endereco=0x7f5e9e1d9030)
...
>
```

A idéia geral do funcionamento foi apresentada acima, mas vamos descer mais um degrau e prosseguir com o detalhamento. Para tal, a explicação sobre o ELF/Linux/AMD64 será dividido em duas partes. A seção 10.3.3.1 detalha as variáveis globais internas à biblioteca compartilhada enquanto que a seção 10.3.3.2 detalha as variáveis globais externas à biblioteca compartilhada.

10.3.3.1 Globais internas à biblioteca compartilhada

Variáveis globais internas à biblioteca são aquelas declaradas dentro de um arquivo objeto de uma biblioteca e só são conhecidas dentro daquele arquivo objeto. O exemplo de trabalho da seção 10.3.1 contém duas variáveis desta categoria: `gA_int` e `gB_int`. Apesar do escopo das variáveis estar associado ao arquivo objeto, iremos utilizar os termos “globais internas à biblioteca compartilhada” para não confundir com as demais variáveis globais.

Como estas variáveis só são utilizadas e visíveis dentro da biblioteca compartilhada então, em tempo de execução, elas devem ser alocados no espaço de memória virtual reservado à biblioteca compartilhada.

Conforme descrito na seção anterior, em tempo de execução será alocada uma região de memória virtual para conter as variáveis globais internas e uma região de memória virtual para conter os apontadores para estas variáveis, chamada `.got`.

O ligador estático deve inserir informações especiais no arquivo objeto compartilhado para auxiliar o carregador e o ligador dinâmico.

A análise que será feita segue o roteiro a seguir:

- indicação do trecho de código contendo `gA_int` ser analisado;
- análise do arquivo objeto;
- análise em tempo de execução;

10.3.3.1.1 Trecho de código analisado

Vamos começar a nossa análise pelo arquivo `a.c` que contém a variável global interna `gA_int`. Vamos analisar o código gerado a partir das linhas 6, 7 e 8 do algoritmo 10.2 copiado abaixo.

```
6 void funcao_interna_em_a (long int x) {
7     gA_int = gA_int + x;
8 }
```

10.3.3.1.2 Análise do arquivo objeto

O código objeto gerado para este trecho é:

```
> gcc -fPIC -c a.c -o a.o
> objdump -d a.o
...
0000000000000000 <funcao_interna_em_a>:
   0:      55                push    %rbp
   1:      48 89 e5          mov     %rsp,%rbp
   4:      48 89 7d f8       mov     %rdi,-0x8(%rbp)
   8:      48 8b 05 00 00 00 00 mov     0x0(%rip),%rax      # f <funcao_interna_em_a+0xf>
   f:      48 8b 10          mov     (%rax),%rdx
  12:      48 8b 45 f8       mov     -0x8(%rbp),%rax
  16:      48 01 c2          add     %rax,%rdx
  19:      48 8b 05 00 00 00 00 mov     0x0(%rip),%rax      # 20 <funcao_interna_em_a+0x20>
  20:      48 89 10          mov     %rdx,(%rax)
  23:      90                nop
  24:      5d                pop     %rbp
  25:      c3                retq
...
```

As instruções nos endereços `0x8` e `0xf` copiam o valor de `gA_int` em `%rdx` conforme o modelo apresentado anteriormente. De forma análoga, as instruções nos endereços `0x19` e `0x20` copiam o resultado da operação de soma de volta para `gA_int`.

Observe que o deslocamento (`movq $(desloc)(%rip), %rax`) é igual a zero. Os zeros estão nos endereços `b`, `c`, `d` e `e` (confira).

10.3.3.1.3 Análise em tempo de execução

Após o carregador ter copiado este código para a memória, ele irá substituir os quatro bytes iguais a zeros do deslocamento para um valor efetivo em tempo de execução. Para auxiliar este processo, o ligador estático inclui a lista de endereços a serem relocados no arquivo objeto, mais especificamente numa estrutura denominada tabela de relocações (aliás, um nome bem apropriado), que pode ser vista com o programa `objdump`:

```
> objdump -r a.o
a.o: formato do arquivo elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET          TYPE          VALUE
000000000000000b R_X86_64_REX_GOTPCRELX gA_int-0x0000000000000004
000000000000001c R_X86_64_REX_GOTPCRELX gA_int-0x0000000000000004
0000000000000035 R_X86_64_REX_GOTPCRELX gA_ext-0x0000000000000004
...
```

A tabela de relocações mostra vários registros, dos quais separamos os três primeiros. Vamos analisar somente o primeiro registro. Ele indica que o conteúdo do *offset* `0xb` deve ser relocado pelo carregador. Indica também que é do tipo relocação da GOT associado à variável `gA_int`, e que devem ser relocados quatro bytes a partir do *offset* `0xb` (ou seja, *offsets* `b`, `c`, `d`, `e`).

Após terem sido alocados os espaços para a `.got` e para a seção `.data`, o carregador executa a relocação para os endereços apropriados, o que pode ser visto com o programa `gdb`.

```
> gdb main
...
(gdb) break funcao_interna_em_a
(gdb) run
Breakpoint 1, ...
(gdb) disassemble funcao_interna_em_a
Dump of assembler code for function funcao_interna_em_a:
    0x00007ffff7bd376a <+0>:      push    %rbp
    0x00007ffff7bd376b <+1>:      mov     %rsp,%rbp
    0x00007ffff7bd376e <+4>:      mov     %rdi,-0x8(%rbp)
    0x00007ffff7bd3772 <+8>:      mov     0x200847(%rip),%rax      # 0x7ffff7dd3fc0
    0x00007ffff7bd3779 <+15>:     mov     (%rax),%rdx
    0x00007ffff7bd377c <+18>:     mov     -0x8(%rbp),%rax
    0x00007ffff7bd3780 <+22>:     add     %rax,%rdx
    0x00007ffff7bd3783 <+25>:     mov     0x200836(%rip),%rax      # 0x7ffff7dd3fc0
    0x00007ffff7bd378a <+32>:     mov     %rdx,(%rax)
    0x00007ffff7bd378d <+35>:     nop
    0x00007ffff7bd378e <+36>:     pop     %rbp
...
(gdb) x/2x 0x7ffff7dd3fc0
0x7ffff7dd3fc0:      0xf7dd4030      0x00007fff
...
(gdb) x/2x 0x7ffff7dd4030
0x7ffff7dd4030 <gA_int>:      0x0000000f      0x00000000
```

Neste caso, a relocação mudou `0x00000000(%rip)` para `0x00200847(%rip)`. O comentário mostra que o alvo (na GOT) está no endereço `0x7ffff7dd3fc0`. O penúltimo comando mostra que `gA_int` reside no endereço `0x00007ffff7dd4030` e o último mostra que o valor de `gA_int` é 15 (`0xf`).

10.3.3.2 Globais da biblioteca compartilhada visíveis externamente

Agora vamos apresentar o que ocorre com as variáveis que são declaradas dentro da biblioteca compartilhada mas que são visíveis fora dela através da diretiva `extern`. No nosso exemplo de trabalho da seção 10.3.1 estas variáveis são `gA_ext` e `gB_ext`, mas iremos trabalhar somente com a primeira.

A variável só pode ser declarada uma única vez sem o modificador `extern`. No exemplo, `gA_ext` é declarada na linha 3 do arquivo `a.c`. Em cada arquivo que utilizar a variável, ela deve ser declarada com o modificador `extern`, como ocorre na linha 3 do arquivo `main.c`.

Conceitualmente, isto significa que o arquivo objeto `a.o` é quem deve reservar espaço na memória para a variável `gA_ext`, mas na prática não é o que acontece. Por exemplo, com bibliotecas estáticas o executável estático agrupa todas as seções `.data` em uma única (veja figura 26), o que significa que ela irá residir na seção `.data` do programa principal.

Já com bibliotecas compartilhadas, a localização depende das opções de compilação.

Relembre que a opção de compilação `-fPIC` (Position-Independent Code) é utilizada para gerar arquivos objeto cujo código pode ser carregado em qualquer posição de memória. Quando um programa é compilado com esta opção, o compilador não utiliza endereçamento fixo nem para variáveis globais e nem para procedimentos, ou seja, utiliza endereçamento relativo para tudo.

Porém, isto se refere aos arquivos objeto compartilhados e não ao programa principal, que continua sendo carregado no segmento de processos. A questão é se isto pode vir a ser um problema de segurança, uma vez que os procedimentos do programa principal (assim como variáveis globais) sempre são carregados para endereços muito próximos cada vez que são executados podendo, em tese, tornar-se alvo de ataques.

Para reforçar o aspecto de segurança, foi incluída uma opção especial de compilação no `gcc` que diz ao compilador ele deve também fazer com que o programa principal seja carregado em qualquer endereço de memória. Esta opção é denominada `PIE` (Position-Independent Executable), que pode ser ativada uma diretiva `-pie` e desativada com a diretiva `-no-pie`. Exemplo:

```

> gcc main.c -lMySharedLib -L. -o main -no-pie
> ./main
Procedimento main: (Endereco=0x400776)
Procedimento a: (Endereco=0x400660)
Procedimento b: (Endereco=0x400630)
Procedimento printf: (Endereco=0x400640)
gA_ext: (Valor=16) (Endereco=0x601050)
gA_int: (Valor=15) (Endereco=0x7fa9287a3030)
funcao_interna_em_a: (Endereco=0x7fa9285a276a)
gB_ext: (Valor=14) (Endereco=0x601048)
gB_int: (Valor=13) (Endereco=0x7fa9287a3040)

> gcc main.c -lMySharedLib -L. -o main -pie
> ./main
Procedimento main: (Endereco=0x55bbddb7e7fa)
Procedimento a: (Endereco=0x7f80493cc790)
Procedimento b: (Endereco=0x7f80493cc82a)
Procedimento printf: (Endereco=0x7f804903fe80)
gA_ext: (Valor=16) (Endereco=0x55bbddd7f018)
gA_int: (Valor=15) (Endereco=0x7f80495cd030)
funcao_interna_em_a: (Endereco=0x7f80493cc76a)
gB_ext: (Valor=14) (Endereco=0x55bbddd7f010)
gB_int: (Valor=13) (Endereco=0x7f80495cd040)

```

É importante destacar que grupo de desenvolvimento do compilador `gcc` não exige o uso de nenhuma das opções, e a escolha fica sob responsabilidade da distribuição Linux adotada.

É possível visualizar esta opção como segue:

```

> gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ... --enable-default-pie ...
gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04.1)

```

Agora retornamos ao tema desta seção: globais da biblioteca compartilhada visíveis externamente.

A análise que segue será feita utilizando a versão com `-pie`, e segue o mesmo roteiro utilizado para globais internas:

- indicação do trecho de código contendo `gA_ext` ser analisado;
- análise do arquivo objeto;
- análise em tempo de execução;

10.3.3.2.1 Trecho de código analisado

A variável `gA_ext` aparece em dois arquivos (`main.c` e `a.c`) que serão carregados em endereços virtuais diferentes em tempo de execução. Para descrever as diferenças de cada caso, a análise será feita sobre os dois arquivos.

Iremos analisar como o acesso à variável é feito em cada arquivo. Do arquivo `a.c`, iremos trabalhar com a linha 11, onde é feita a atribuição de um valor para a variável.

```
11 gA_ext = 16;
```

No arquivo `main.c`, o trecho a ser analisado é a linha 13, onde também é feita uma atribuição para aquela variável.

```
13 gA_ext = 15;
```

10.3.3.2.2 Análise do arquivo objeto

O conteúdo dos arquivos objeto que correspondem aos trechos de código analisados são listados a seguir.

```
> objdump -d a.o
...
32:      48 8b 05 00 00 00 00      mov     0x0(%rip),%rax      # 39 <a+0x13>
39:      48 c7 00 10 00 00 00      movq    $0x10, (%rax)
...
> gcc -c main.c -o main.o
> objdump -d main.o
...
7fb:      48 89 e5                  mov     %rsp,%rbp
7fe:      48 c7 05 0f 08 20 00      movq    $0xf,0x20080f(%rip)  # 201018 <gA_ext>
...
```

Após a análise feita sobre variáveis globais internas, o leitor já deve ter percebido que o arquivo `a.o` acessa a variável global externa através da GOT enquanto que o arquivo objeto `main.o` acessa a mesma variável utilizando o modo de endereçamento indireto via `%rip` (veja exercício 10.9).

A análise destes dois arquivos sugere que, em tempo de execução, a variável `gA_ext` estará localizada na seção `.data` próxima ao local onde o procedimento `main` será alocado.

10.3.3.2.3 Análise em tempo de execução

A análise do endereço e da forma de acessar a variável `gA_ext` em tempo de execução pode ser feita com o auxílio do `gdb` conforme resumido abaixo:

```
> gdb main
(gdb) break main
(gdb) run
(gdb) disassemble main
...
0x00005555555547fe <+4>:      movq    $0xf,0x20080f(%rip)      # 0x555555755018 <gA_ext>
...
(gdb) disassemble a
...
0x00007ffff7bd379c <+12>:      mov     0x200855(%rip),%rax      # 0x7ffff7dd3ff8
0x00007ffff7bd37a3 <+19>:      movq    $0x10, (%rax)
...
(gdb) x/2x 0x7ffff7dd3ff8
0x7ffff7dd3ff8:      0x55755018      0x00005555
...
```

O extrato mostra que, nesta execução, o endereço da variável `gA_ext` é `0x555555755018`. O procedimento `main()` a acessa através de endereçamento indireto via `%rip`. Já no caso do procedimento `a()`, a instrução utilizada primeiro acessa a got no endereço `0x7ffff7dd3ff8`, cujo conteúdo é `0x555555755018`, o endereço da variável `gA_ext`.

10.3.4 Acesso exclusivo às variáveis do processo

Como foi visto no capítulo 4, o espaço virtual de cada processo não pode ser acessado por outro processo. Porém, quando se trata de bibliotecas compartilhadas, as coisas ficam um pouco nebulosas já que existe uma região que dois ou mais processos podem acessar.

O problema decorrente do compartilhamento é fundamentalmente uma questão de segurança pois as informações críticas de um processo não podem ser acessadas por nenhum outro processo. Sendo assim, vamos primeiro analisar quais são as informações contidas em um processo e quais delas são críticas. Para esta análise, vamos utilizar a figura 32, Uma reimpressão da figura 28.

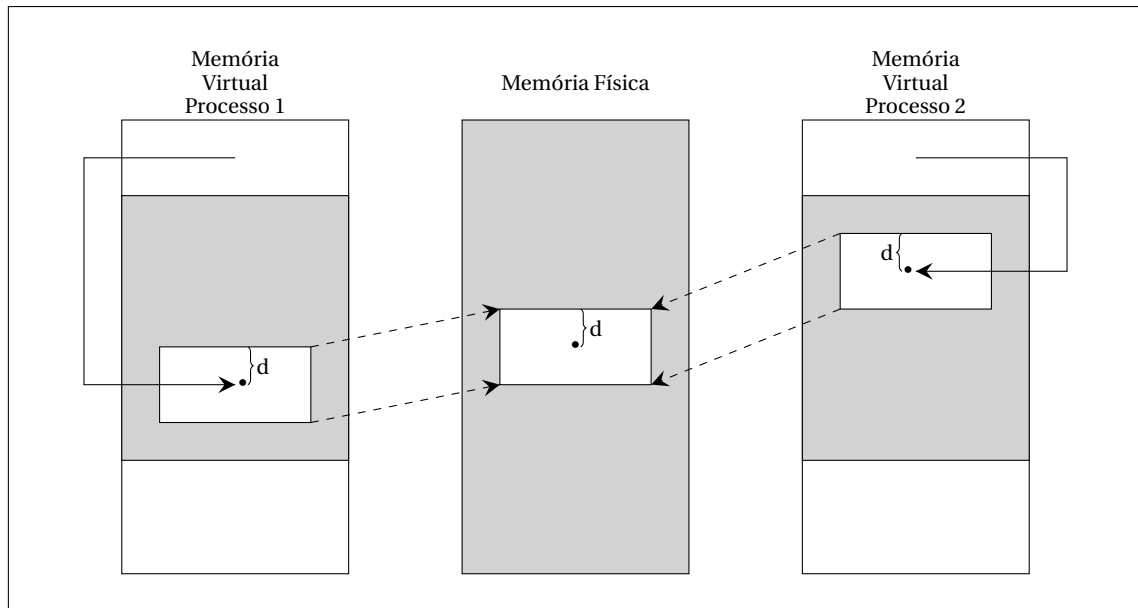


Figura 32 – Biblioteca compartilhada mapeada em dois processos.

A parte superior da memória virtual de cada processo representa o segmento de processo, onde há o espaço reservado às seções `.text`, `.data`, `.bss`, `heap` e pilha, assuntos da primeira parte do livro. Estas regiões não são compartilhadas, ou seja, são de acesso exclusivo do processo em questão. Assim, se o fluxo de execução estiver no segmento de processo do processo 1, ele não poderá acessar nenhuma informação do segmento de processo do processo 2.

Agora considere a execução na região compartilhada. Esta região também tem seção `.text` (instruções), seção `.data` e seção `.bss` (variáveis globais), porém não existe o espaço para a `heap` nem para a pilha. Para entender porque, suponha que o processo 1 inicia a execução da seção `.text` da biblioteca compartilhada. Vamos analisar o que ocorre se o processo precisar utilizar a `heap` ou a pilha.

Vamos começar pela `heap`. Suponha que a biblioteca compartilhada execute a função `malloc()`. Conforme discutido no capítulo 6, o limite da `heap` é determinado pela variável `brk`, sendo que só uma variável `brk` por processo. Isto significa que cada processo só tem uma região reservada à `heap`, e já que esta região está mapeada no segmento de processo não há risco das informações armazenadas na `heap` pela biblioteca compartilhada sejam acessadas por qualquer outro processo.

Agora vamos analisar uma chamada de procedimento disparada dentro da biblioteca compartilhada pelo processo 1. Conforme descrito no capítulo 3, a chamada de um procedimento implica na criação de um registro de ativação e o acesso às informações de um registro de ativação é feito em função do registrador `%rbp`. Como o valor desse registrador é individual em cada processo, o registro de ativação da chamada de procedimentos do processo 1 é criado na região da pilha do segmento de processo do processo 1, com acesso exclusivo do processo 1. Em outras palavras: as informações contidas nos registros de ativação relacionadas com chamadas de procedimento de bibliotecas compartilhadas não são compartilhadas.

Agora, vamos tratar das seções `.text`, `.data` e `.bss` da biblioteca compartilhada.

Compartilhar a seção `.text` não é propriamente um problema, pois ela contém somente instruções e não pode ser modificada. O único problema que pode ocorrer está relacionado com um código secreto. Neste caso não é recomendado o uso de bibliotecas compartilhadas. Para ser sincero, também não é adequado o uso de bibliotecas estáticas pelo mesmo motivo.

O problema ocorre com as seções do objeto compartilhado que armazenam variáveis globais como as seções `.data` e `.bss`, já que elas podem residir no objeto compartilhado e, em princípio, poderiam ser acessadas por todos os processos que a usam, algo perigoso pois podem conter informações sigilosas.

O objetivo é fazer com as variáveis globais o mesmo que foi feito com as variáveis da pilha e da `heap`: colocar estas variáveis em regiões com acesso individualizado e exclusivo por processo.

Para tal, a solução reside na paginação. Para explicar como a paginação resolve este problema, iremos utilizar a figura 33. A figura mostra dois processos em execução em suas respectivas memórias virtuais e o trecho que é compartilhado entre as duas na memória física. A memória virtual exclusiva de cada um dos dois processos está destacada com

padrões diferentes exatamente para sugerir a exclusividade. Isto quer dizer que qualquer variável armazenada na região destacada do processo 1 não pode ser acessada pelo processo 2 e vice-versa. Isto é implementado pelo mecanismo de paginação, onde as páginas de um processo não podem ser acessadas por qualquer outro.

Agora, observe a região compartilhada dos dois processos e sua representação na memória física, mais especificamente os dois quadrados no objeto compartilhado. A ideia é que nestes dois quadrados estão as variáveis globais mapeadas em páginas. Elas podem representar a seção `.data`, `.got`, `.plt.got`, `.bss` ou outras, mas para a explicação que segue, considere que uma delas representa a seção `.data` e a outra representa a seção `.got`.

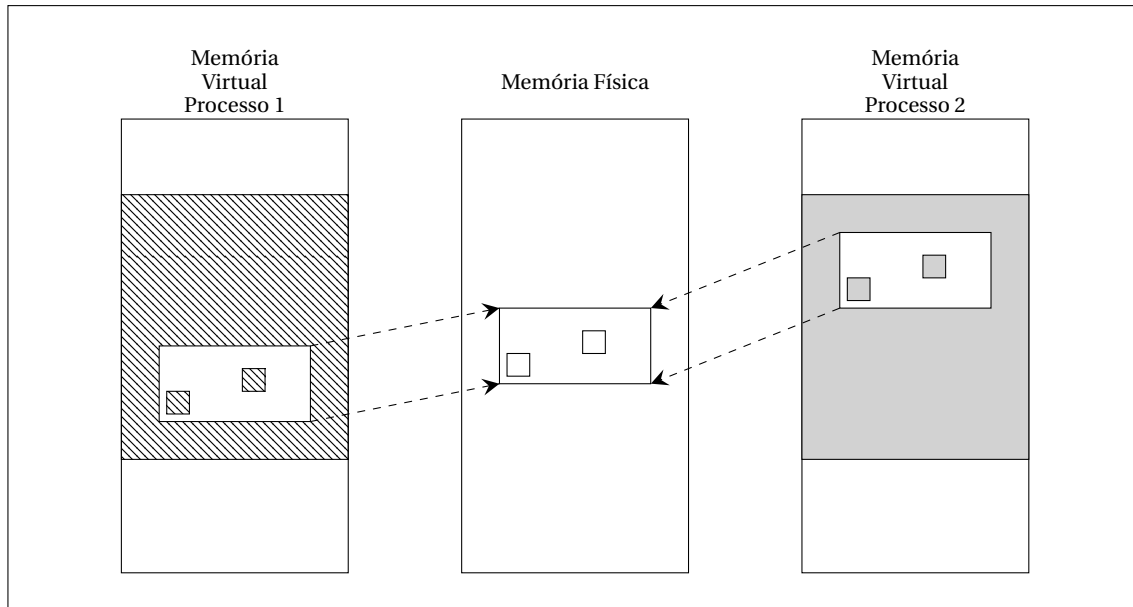


Figura 33 – Acesso exclusivo às variáveis do processo usando paginação.

Cada quadrado está representado também nas memórias virtuais de cada processo. Observe que em cada memória virtual o quadrado está representado pela cor que representa as páginas cujo acesso é exclusivo daquele processo.

Para exemplificar o funcionamento, considere que o processo 1 está executando o trecho de código do objeto compartilhado. Vamos analisar o que acontece na execução de uma instrução que acessa uma variável global através da `.got`. Como já sabemos da seção 10.3.3, esta operação é feita em duas instruções assembly, por exemplo:

```
0x00007ffff7bd379c <+12>:      mov     0x200855(%rip),%rax      # 0x7ffff7dd3ff8
0x00007ffff7bd37a3 <+19>:      movq    $0x10, (%rax)
```

A primeira instrução está no endereço virtual `0x7ffff7bd379c` do processo 1. É uma região compartilhada pelos os dois processos, ou seja, os dois processos “enxergam” exatamente a mesma instrução, só que em endereços virtuais diferentes.

Esta instrução acessa o endereço virtual distante `0x200855` bytes da instrução corrente, ou seja, o endereço virtual `0x7ffff7dd3ff8`¹⁰ do processo 1. Em nosso exemplo, este endereço está mapeado em um dos quadrados da memória física, para o qual foi alocada uma página virtual exclusiva no processo 1. Isto está representado no lado esquerdo da figura 33 com o quadrado hachurado.

O endereço (`0x7ffff7dd3ff8`) está contido na `.got`, e como já sabemos, ele contém o endereço da variável global. Como exemplo, considere que `0x555555755018` é o endereço da variável global (ou seja, `M[0x7ffff7dd3ff8] = 0x555555755018`). Este endereço está localizado na outra caixa destacada na figura 33.

Agora suponha que o processo 2 executa o mesmo para de instruções assembly. O local da `got` na memória física é o mesmo quadrado acessado pelo processo 1. A diferença é que o quadrado cinza do lado direito é que será utilizado, ou seja uma região de acesso exclusivo do processo 2.

Desta forma, a paginação permite que os processos compartilhem a mesma página física no caso de páginas só de leitura (como por exemplo, instruções) e separa as páginas de escrita como exclusivas de cada processo. Isto também implica dizer que o carregador deve mapear a seção `.data` e a seção `.text` em páginas diferentes.

¹⁰ `0x7ffff7bd37a3 + 0x200855 = 0x7ffff7dd3ff8`

10.3.4.1 Visualizando a paginação no linux

Para finalizar este capítulo, vamos descer perigosamente mais um nível a camada de abstração do Linux e mostrar como o mapeamento de memória virtual de um programa que utiliza biblioteca compartilhada, chegando ao ponto de examinar as permissões das páginas de memória virtual alocados para uma biblioteca compartilhada.

Para atingir este objetivo, iremos utilizar as informações disponibilizadas pelo sistema no diretório `/proc`. Este diretório contém informações sobre quase tudo que o sistema operacional conhece do *hardware* no computador como por exemplo quanto espaço de memória física existe (`/proc/meminfo`), informações sobre a CPU (`/proc/cpuinfo`), entre muitas outras.

Uma das coisas interessantes sobre este diretório é que ele não está mapeado em disco, sendo conhecido unicamente em memória física (utilize o comando `du /proc` para comprovar). Por esta razão, ele também é chamado de diretório virtual ou também de pseudo-diretório.

Além de todas essas informações sobre o computador, o diretório `/proc` também armazena informações sobre cada processo em execução no sistema como por exemplo a linha de comando utilizada para executar aquele processo, descritores de arquivos utilizados, entre muitos outros.

Estas informações pode ser visualizadas ao saber o número de identificação do processo (PID). Com esta informação, basta um olhar no diretório `/proc/<PID>`. Por exemplo, para o processo com PID=22998, temos:

```
> ls /proc/22998/
attr          exe           mounts        projid_map    status
autogroup     fd           mountstats    root          syscall
auxv          fdinfo       net           sched         task
cgroup        gid_map      ns            schedstat     timers
clear_refs    io           numa_maps     sessionid     timerslack_ns
cmdline       limits       oom_adj        setgroups     uid_map
comm          loginuid     oom_score      smaps         wchan
coredump_filter map_files    oom_score_adj smaps_rollback
cpuset        maps         pagemap       stack
cwd           mem          patch_state    stat
environ       mountinfo    personality     statm
```

Um dos arquivos listados é o arquivo `maps`, que contém informações sobre o mapa da memória virtual em tempo de execução, e é exatamente com este arquivo iremos trabalhar nesta seção.

Para tal, fizemos na modificação no nosso programa de trabalho conforme pode ser visualizado no algoritmo 10.7. Este novo arquivo insere uma funcionalidade importante para a visualização: o número de identificação de processo, PID, que ele imprime na linha 25. Além disto, ele aguarda interação do usuário pelo comando na linha 26, o que permite a exploração das informações no diretório `/proc` enquanto o usuário não interagir.

Ao executar o programa, obtive o seguinte resultado :

```
> ./main-pid
Procedimento main: (Endereco=0x55890bc7688a)
Procedimento a: (Endereco=0x7f090c700790)
Procedimento b: (Endereco=0x7f090c70082a)
Procedimento printf: (Endereco=0x7f090c373e80)
gA_ext: (Valor=16) (Endereco=0x55890be77018)
gA_int: (Valor=15) (Endereco=0x7f090c901030)
funcao_interna_em_a: (Endereco=0x7f090c70076a)
gB_ext: (Valor=14) (Endereco=0x55890be77010)
gB_int: (Valor=13) (Endereco=0x7f090c901040)
PID=9316
```

A última linha mostra o número do processo: 9316. Sabendo o número do processo, vamos analisar o conteúdo do arquivo com o comando abaixo:

```
> cat /proc/9316/maps
55890bc76000-55890bc77000 r-xp 00000000 00:34 93257763    .../main-pid
55890be76000-55890be77000 r--p 00000000 00:34 93257763    .../main-pid
55890be77000-55890be78000 rw-p 00001000 00:34 93257763    .../main-pid
55890c4ee000-55890c50f000 rw-p 00000000 00:00 0          [heap]
7f090c30f000-7f090c4f6000 r-xp 00000000 00:19 33         /lib/x86_64-linux-gnu/libc-2.27.so
```

```

1  #include <stdio.h>
2  #include <unistd.h>
3
4  extern long int gA_ext;
5  extern long int gB_ext;
6
7  long int gMain;
8
9  void a (char* s);
10 void b (char* s);
11
12 long int main ()
13 {
14     gA_ext=15;
15     gB_ext=14;
16
17     printf ("Procedimento main: (Endereco=%p)\n", main);
18     printf ("Procedimento a: (Endereco=%p)\n", a);
19     printf ("Procedimento b: (Endereco=%p)\n", b);
20     printf ("Procedimento printf: (Endereco=%p)\n", printf);
21
22     a("dentro de a\n");
23     b("dentro de b\n");
24
25     printf ("PID=%d\n", getpid());
26     getchar();
27
28 }

```

Algoritmo 10.7 – Arquivo main.c

```

7f090c4f6000-7f090c6f6000 ---p 001e7000 00:19 33      /lib/x86_64-linux-gnu/libc-2.27.so
7f090c6f6000-7f090c6fa000 r--p 001e7000 00:19 33      /lib/x86_64-linux-gnu/libc-2.27.so
7f090c6fa000-7f090c6fc000 rw-p 001eb000 00:19 33      /lib/x86_64-linux-gnu/libc-2.27.so
7f090c6fc000-7f090c700000 rw-p 00000000 00:00 0
7f090c700000-7f090c701000 r-xp 00000000 00:34 93258334  .../libMySharedLib.so
7f090c701000-7f090c900000 ---p 00001000 00:34 93258334  .../libMySharedLib.so
7f090c900000-7f090c901000 r--p 00000000 00:34 93258334  .../libMySharedLib.so
7f090c901000-7f090c902000 rw-p 00001000 00:34 93258334  .../libMySharedLib.so
7f090c902000-7f090c929000 r-xp 00000000 00:19 30      /lib/x86_64-linux-gnu/ld-2.27.so
7f090cad4000-7f090cad7000 rw-p 00000000 00:00 0
7f090cb27000-7f090cb29000 rw-p 00000000 00:00 0
7f090cb29000-7f090cb2a000 r--p 00027000 00:19 30      /lib/x86_64-linux-gnu/ld-2.27.so
7f090cb2a000-7f090cb2b000 rw-p 00028000 00:19 30      /lib/x86_64-linux-gnu/ld-2.27.so
7f090cb2b000-7f090cb2c000 rw-p 00000000 00:00 0
7ffdd25e4000-7ffdd2605000 rw-p 00000000 00:00 0      [stack]
7ffdd2720000-7ffdd2723000 r--p 00000000 00:00 0      [vvar]
7ffdd2723000-7ffdd2725000 r-xp 00000000 00:00 0      [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0      [vsyscall]

```

A impressão é difícil de compreender. A ideia geral é que a primeira coluna indica a faixa de endereços virtuais, a última coluna indica o que foi colocado naquela página virtual, e ao centro estão impressas as permissões das páginas virtuais. Como exemplo, veja que a primeira linha tem permissão `r-xp`, o que indica que aquele conjunto de páginas pode ser lido, não pode ser escrito e pode ser executado. A última coluna contém o caractere "p", que indica que aquela página é privativa daquele processo. No caso de bibliotecas compartilhadas, isto significa que enquanto o processo utilizar aquela página unicamente para leitura, a página será mapeada para mesma cópia em memória física que todos os demais processos utilizam. Caso o processo escreva algo naquela página, só então será criada uma página virtual específica para aquele processo. Este mecanismo é chamado de *copy on write* e recai sobre a categoria de mecanismos de otimização descrita ao final da seção 10.2.1.2, chamados de *lazy*.

A tabela 19 é uma reimpressão sintetizada daquela listagem onde o tamanho de cada objeto (ou espaços vazios

entre eles). Alguns pontos se destacam:

End. Virtuais		Tamanho (em páginas)	Conteúdo
Início	Fim		
0x55890bc76000	0x55890be78000	514	main-pid
0x55890be78000	0x55890c4ee000	1.654	— vazio —
0x55890c4ee000	0x55890c50f000	33	heap
0x55890c50f000	0x7f090c30f000	11.140.070.912	— vazio —
0x7f090c30f000	0x7f090c6fc000	1.005	libc-2.27.so
0x7f090c6fc000	0x7f090c700000	4	— vazio —
0x7f090c700000	0x7f090c902000	514	libMySharedLib.so
0x7f090c902000	0x7f090c929000	39	ld-2.27.so
0x7f090c929000	0x7f090cb29000	512	— vazio —
0x7f090cb29000	0x7f090cb2b000	2	ld-2.27.so
0x7f090cb2b000	0x7f090cb2c000	1	— vazio —
0x7ffdd25e4000	0x7ffdd2605000	33	[stack]
0x7ffdd2720000	0x7ffdd2723000	3	[vvar]
0x7ffdd2723000	0x7ffdd2725000	2	[vdso]
0xfffffffff600000	0xfffffffff601000	1	[vsyscall]

Tabela 19 – Memória virtual do programa 10.7 em tempo de execução

- A tabela indica que foram reservadas 33 páginas para a *heap* e outras 33 páginas para a pilha;
- a biblioteca compartilhada *libc-2.27.so* foi mapeada em um conjunto de páginas consecutivas (está toda em um único lugar);
- o ligador dinâmico *ld-2.27.so* foi dividido em dois lugares diferentes;
- ao final da impressão estão listados alguns nomes estranhos: *vvar*, *vdso* e *vsyscall*. Estas páginas são utilizadas como pontos de entrada para acelerar certas chamadas de sistema que podem ser executadas no modo usuário como por exemplo `gettimeofday()`. Originalmente estes pontos de entrada eram localizados em endereços fixos, o que poderia ser um ponto vulnerável do sistema. Por esta razão, foi criada uma espécie de trampolim cujos endereços de entrada são aleatórios (*vdso*) o que fortalece segurança do sistema. Para mais detalhes veja [9] e `man vdso`.

Agora vamos descer mais um degrau e analisar as páginas alocadas para a biblioteca compartilhada *libMySharedLib.so* em tempo de execução. Observe que na listagem do mapa de memória virtual do processo, quatro linhas são dedicadas à biblioteca compartilhada *libMySharedLib.so*.

Copiamos estas quatro linhas para a tabela 20 e lá acrescentamos algumas informações. A primeira linha da tabela mostra que foi alocada uma página virtual com permissão *r-xp*, ou seja, de leitura (*r*) e execução (*x*), mas não de escrita (*w*). Ao combinar o resultado da execução do programa com a faixa de endereços desta página, é possível deduzir que esta página contém a seção *text* da biblioteca, o que pode ser comprovado ao observar que os endereços dos procedimentos `a()` e `b()` encontram-se nesta página.

End. Virtuais		Tamanho (em páginas)	Permissões	Conteúdo
Início	Fim			
0x7f090c700000	0x7f090c701000	1	r-xp	seção .text a() : 0x7f090c700790 b() : 0x7f090c70082a
0x7f090c701000	0x7f090c900000	511	---p	— vazio —
0x7f090c900000	0x7f090c901000	1	r--p	seção .got
0x7f090c901000	0x7f090c902000	1	rw-p	seção .data gA_int : 0x7f090c901030 gB_int : 0x7f090c901040

Tabela 20 – Permissões das páginas da biblioteca *libMySharedLib.so* em tempo de execução

Observe que as permissões desta página são coerentes com o que é esperado na seção `.text`. A segunda linha mostra que foram mapeadas 511 páginas vazias (veja exercício 10.10). A terceira linha indica a existência de uma única página com permissão somente de leitura. Ao analisar a execução do programa dentro do `gdb`, foi possível identificar que esta página contém os apontadores da `got`. A quarta e última linha indica a presença de uma página logo após a `got` com permissões de leitura e de escrita, mas não de execução. Ao comparar os endereços desta página com o resultado da execução do programa, é possível localizar os endereços das variáveis globais `gA_int` e `gB_int` e assim deduzir que esta página contém a seção `.data`.

Exercícios

- 10.1 Diferente do AMD64, o seu precursor de 32 bits (x86) não permite incluir o *instruction pointer* no endereçamento indireto (ou seja, não permite instruções do tipo `movq <desloc>(%rip), %rax`. Porém, encontra-se código como:

```
call r01
r01: pop %eax
```

Explique como isto¹¹ pode ser usado para implementar o mecanismo de acesso às variáveis locais internas e externas da biblioteca.

- 10.2 O número de entradas no vetor `.got.plt` é calculado pelo ligador estático. Proponha um algoritmo que o ligador estático poderia executar para fazê-lo.
- 10.3 Na seção 10.3.1 é levantada a questão da localização (endereço em tempo de execução) das variáveis globais internas e externas ao programa principal quando é utilizada uma biblioteca compartilhada. Qual o endereço destas variáveis globais em bibliotecas estáticas? Responda esta pergunta e depois recompile o programa com biblioteca estática. Analise a resposta e desenhe a localização delas. Explique por que é lá.
- 10.4 Considere uma biblioteca uma única variável global que é acessada n vezes na região de código da biblioteca. Compare o desempenho da abordagem apresentada na seção 10.3.3 com o desempenho atualizando os n *fixups*. Para qual valor de n a abordagem de *fixups* passa a ser mais cara? Como o cálculo exato é complicado, use uma abordagem generalista.
- 10.5 Explique por que não se utiliza o endereço absoluto para acessar a `GA_ext` a partir da seção `.text`.
- 10.6 usando o comando `objdump -S a.o`, é possível observar o código gerado pelo arquivo `a.c` (veja seção 10.3.1). Porém, é curioso que a linha 16 do algoritmo 10.2 (`gA_ext=16;`) tenha sido traduzida para:

```
....
32:      48 8b 05 00 00 00 00      mov     0x0(%rip),%rax      # 39 <a+0x13>
39:      48 c7 00 10 00 00 00      movq    $0x10, (%rax)
....
```

A presença de `0x0(%rip)` surpreende? Era o esperado após o que foi apresentado na seção 10.3.3? Aproveite o exemplo e diferencie arquivos objeto (gerados com `-fPIC`) de bibliotecas compartilhadas.

- 10.7 Suponha que a variável `gA_ext` seja declarada como `extern` no arquivo `b.c` (algoritmo 10.3) e nele seja atribuído um valor. Em tempo de execução será incluída uma nova entrada na `.got` do espaço virtual de `b.o` ou aquela de `a.o` será reaproveitada? E se `b.o` estiver em outra biblioteca?
- 10.8 Recompile os programas da seção 10.3.1 criando uma biblioteca estática. Para gerar o executável, use a opção `-static`, e analise a impressão. Os endereços das variáveis e das funções é estático ou aleatório? Correspondem a endereços dentro da área de segmentos de processo ou de segmentos de objetos compartilhados. Explique.
- 10.9 Na análise do arquivo objeto das variáveis globais da biblioteca compartilhada visíveis externamente (página 163) o arquivo `main.o` foi gerado na versão estática (sem a opção `-fPIC`). Quais as diferenças no arquivo objeto `main.o` e no arquivo executável `main` se fosse utilizada a opção `-fPIC`? Explique.
- 10.10 A tabela 20 apresenta 511 páginas entre a seção `.text` e a `got`. Executei o programa várias vezes e sempre existem 511 páginas entre a seção `.text` e a `got`. Explique porque (dica: use o `objdump`).
- 10.11 Onde fica mapeada a `.got.plt` na tabela 20? Justifique e comprove no `gdb`.

¹¹ esta gambiarra

- 10.12 As várias execuções do programa `main` na página 159 mostram que `gA_ext` e `gB_ext` são sempre mapeados para endereços que terminam com `0x018` e `0x030` respectivamente. É coincidência?
- 10.13 O registro de ativação da primeira chamada de um procedimento através da `plt` é muito curioso, pois contém os parâmetros e endereço de retorno do procedimento e os parâmetros do `resolver`.
- Explique porque o `resolver()` não confunde parâmetros do procedimento com os parâmetros do próprio `resolver()`;
 - Explique porque o procedimento não confunde parâmetros do próprio procedimento com os parâmetros do `resolver()`;
 - Desenhe o registro de ativação da chamada do procedimento `proc()` nas seguintes situações:
 - a) primeira chamada de procedimento, imediatamente após entrar em `proc@plt`;
 - b) primeira chamada de procedimento, imediatamente antes de desviar o fluxo para o `resolver()`;
 - c) segunda chamada de procedimento em diante, imediatamente após entrar em `proc@plt`;
 - d) Existe algum truque para retornar o fluxo para a instrução seguinte a `call proc@plt` tanto na primeira chamada quanto da segunda chamada para frente?

11. Interpretadores

Este capítulo trata de programas capazes de executar outros programas. O termo genérico para eles é interpretador, mas é comum utilizar outros termos, em especial emuladores e simuladores.

Formalmente, os três termos referem-se a aplicativos com finalidades diferentes:

Emulador Um emulador é um programa que imita o comportamento de um *hardware* ou aplicativo básico em tempo real. Ele é capaz de duplicar o comportamento das funções de um sistema “A” em um sistema “B” de tal forma que o sistema “B” se comporte exatamente como se fosse o sistema “A”. Em princípio, um emulador pode substituir (se fazer passar por) o *hardware* ou *software* emulado. Exemplos:

- Emulador do Windows em linux;
- Emulador de um console de jogos em um computador pessoal;

Simulador Um simulador é um programa que simula o comportamento de algo real usando fórmulas matemáticas. Assim, enquanto um emulador pode substituir o “alvo”, um simulador não pode. Por exemplo, um simulador de voo não pode transportar uma pessoa do ponto “A” para um ponto “B”. Se pudesse ele seria um emulador de voo¹.

Interpretador O termo interpretador é normalmente associado a um aplicativo capaz de executar um programa escrito em uma linguagem de programação sem gerar código executável na máquina nativa.

O uso mais comum destes termos é: “linguagem de programação interpretada”, “emulador de *hardware*” e “simulador de ambiente”. A Seção 11.1 detalha como funcionam os emuladores enquanto que a Seção 11.2 detalha como funcionam os interpretadores. Não será explicado como funcionam simuladores por se tratar de um mecanismo de modelagem de um fenômeno (muitas vezes real) através de fórmulas matemáticas, que não são o alvo deste livro.

11.1 Emuladores

O termo emulador foi cunhado para indicar um *hardware* que se comporta como outro. Porém, com o tempo, o termo foi estendido para descrever sistemas de aplicativos básicos também. Um exemplo interessante é o wine² (Wine Is Not Emulator - um emulador de aplicações windows para linux).

Este programa lê um arquivo executável no formato usado pelo windows (COM, EXE, PE) e converte para o formato ELF. Após esta conversão, o programa é colocado em execução em linux. A ideia é simples: a seção de instruções do formato origem é copiada para seção “text” do ELF. A seção de dados globais é copiada para a seção “data”, e assim por diante. Com isso, o formato original é totalmente convertido para ELF e o programa pode executar.

Porém, ainda resta um problema. Em tempo de execução, um programa pode (e deve) usar chamadas ao sistema. O programa origem foi projetado para fazer uso das chamadas de sistema do Windows (o equivalente ao POSIX do linux), utilizando parâmetros diferentes nos registradores.

Assim, o wine não converte simplesmente arquivos windows para o formato executável ELF, mas inclui sub-rotinas que emulam as chamadas de sistema do Windows em linux. A ideia trocar chamadas de sistema por chamadas a procedimentos. Se necessário, estas chamadas de procedimento usam as chamadas de sistema nativas do linux.

A versão da API do Windows projetada para o wine não é uma cópia do original (que não é pública), mas sim uma versão livre baseada em engenharia reversa. Para programas bem comportados (ou seja, os que só acessam o *hardware* através das chamadas ao sistema), ele funciona muito bem. Porém para programas que acessam o *hardware* diretamente, não funciona, evidentemente.

O grande problema é como converter gráficos complexos (por exemplo simulações e jogos) do windows para linux. A ideia natural é usar uma biblioteca gráfica para fazer o trabalho, porém como as APIs das bibliotecas gráficas da Microsoft (DirectX) e do linux (OpenGL) não são totalmente compatíveis, podem causar perda de desempenho (ou não funcionar).

¹ <http://stackoverflow.com/questions/1584617/simulator-or-emulator-what-is-the-difference>

² <http://www.winehq.org>

11.2 Interpretadores

Um interpretador é um programa que executa instruções escritas em uma linguagem de programação (ou *script*) sem gerar arquivo executável para a arquitetura em que será executado.

Um interpretador normalmente usa uma das seguintes abordagens³:

1. executar direto do código-fonte. As primeiras versões de LISP⁴ e Basic funcionavam assim;
2. traduzir o código fonte para uma representação intermediária dinâmica e executar esta representação. Perl, Python, Ruby, Matlab entre outras usam esta abordagem;
3. traduzir o código fonte para uma representação intermediária que é salva em arquivo e que depois será executada. Java utiliza esta abordagem.

A seguir vamos detalhar cada abordagem.

11.2.1 Primeira Abordagem

A primeira abordagem para implementar um interpretador é fazê-lo trabalhar com uma linha do código fonte por vez: o interpretador lê a linha i , analisa o que ela faz e a interpreta. Se não implicar em mudança de fluxo, lê a linha seguinte ($i+1$), a analisa e a interpreta, e assim por diante.

O tempo de execução desta categoria de interpretadores é maior uma vez que uma mesma linha do código precisa ser reconhecida, analisada e interpretada todas as vezes que for executada. Por exemplo, considere a interpretação de um comando de atribuição. Para simplificar, suponha que a atribuição é o único comando da linha. Então, o interpretador deve realizar as seguintes tarefas:

1. ler o comando e identificar se é condicional, repetitivo, atribuição, chamada de procedimento, etc. Neste caso, é de atribuição. É importante destacar que esta tarefa é custosa pois envolve operações de busca e comparação de *strings*.
2. verificar se o parâmetro origem já foi declarado. Se não foi, indicar um erro. Se já foi, encontrar o local de memória em que foi alocado;
3. verificar se o parâmetro destino já foi declarado. Se foi, encontrar o local de memória em que foi declarado. Se não foi, criá-lo em algum local de memória;
4. Copiar o conteúdo do parâmetro origem para o parâmetro destino;

Os passos acima são executados para cada comando. Para entender porque interpretadores que usam esta abordagem são lentos, imagine que o comando de atribuição está incluído em um laço. As operações indicadas nas Tarefas 1, 2 E 3 terão que ser repetidas a cada nova iteração obtendo sempre o mesmo resultado para que a Tarefa 4, aquela que realmente importa, possa ser executada.

Esta análise pode ser estendida ao programa inteiro. Quando qualquer instrução for interpretada haverá o custo computacional adicional das Tarefas 1, 2 E 3 para a execução da Tarefa 4.

Uma linguagem que usa esta abordagem é a linguagem `bash`, o que se justifica pois os programas executados ali são normalmente pequenos.

Usaremos um programa escrito em `bash` para demonstrar outro problema com esta abordagem: a verificação sintática só é feita nos comandos executados. Como exemplo, considere o programa `bash` do Algoritmo 11.1. Há um erro sintático na linha 9, que só será detectado se o fluxo de execução passar por aquela linha, e para piorar, o fluxo continua depois do erro como demonstrado a seguir.

```
> bash if.sh 100
101
> bash if.sh 102
if.sh: linha 9: count: comando não encontrado
102
```

Apesar de ineficiente, esta primeira abordagem é muito utilizada em linguagens de programação cujos programas são pequenos.

³ A fonte aqui é a wikipedia ([https://en.wikipedia.org/wiki/Interpreter_\(computing\)](https://en.wikipedia.org/wiki/Interpreter_(computing))), que nem sempre é confiável, mas com a qual eu concordo neste caso.

⁴ Acrônimo para LISP Processing. Como a linguagem usa muitos parênteses, ganhou outro significado: Lots of Infernal Stupid Parenthesis

```
1  #!/bin/bash
2
3  count=$1
4
5  if [ $count -eq 100 ]
6  then
7      count=$(( $count + 1 ))
8  else
9      count = $count - 1
10 fi
11
12 echo $count
```

Algoritmo 11.1 – Programa bash com erro no else

11.2.2 Segunda Abordagem

A segunda abordagem envolve duas fases. Na primeira fase, o programa é convertido em uma estrutura dinâmica antes de começar a execução, por exemplo, um grafo onde cada nó é uma instrução e arestas apontam para a próxima instrução.

Cada nó contém informações próprias da instrução que ela representa. Por exemplo, um nó que corresponde ao comando IF deve conter um ponteiro para a expressão, um nó para indicar o fluxo do THEN e um para o fluxo do ELSE. Visto desta forma, cada nó do grafo assemelha-se a um comando em linguagem de máquina e o interpretador executa essencialmente as mesmas ações que uma CPU em um computador.

Esta primeira fase envolve conceitos utilizados em compiladores, como análise léxica, sintática, semântica e geração de código. A ideia geral é converter o programa fonte em uma árvore sintática abstrata⁵ e acrescentar arestas indicando o fluxo de execução, transformando a árvore em um grafo. O processo é descrito em livros de compiladores e foge ao escopo deste livro.

É importante mencionar que na primeira fase, cada nó do grafo pode apontar diretamente para as variáveis (ou objetos) utilizados naquela instrução. Por exemplo, em uma atribuição, o nó pode ter três informações: a instrução (atribuição), o endereço em memória do operando de origem e o endereço de memória do operando de destino.

É na segunda fase que é feita a execução. Aqui o interpretador seguirá o fluxo indicado pelos nós e quando encontrar o nó correspondente à atribuição, encontrará também o endereço dos operandos.

Ao contrário do que ocorre na primeira abordagem, não é necessário ler a linha para identificar a instrução e localizar os operandos, tornando a execução mais rápida.

As linguagens que utilizam esta abordagem são principalmente aquelas que são orientadas a objeto. Os motivos são vários, mas a explicação exige conhecimento de conceitos de orientação a objetos.

Apesar de fugir ao escopo deste livro, talvez seja possível explicar usando o conceito de herança. Na herança, uma classe derivada herda as características de outra classe, denominada classe pai. Quando o objeto de uma classe derivada é instanciado, uma estrutura de dados é alocada para aquele objeto. Esta estrutura contém referências a todas as variáveis e métodos da classe derivada e uma referência à classe base.

Agora suponha que o objeto invoca um método (um procedimento). O interpretador procura o método dentro da classe derivada, mas se não encontrar lá procura na classe pai. Como a classe pai também pode ser uma classe derivada de outra classe, procura na classe avô e a busca continua até encontrar o método ou verificar em toda a hierarquia de classes.

A explicação acima é uma simplificação grosseira, pois apresenta um algoritmo muito lento que pode (e é) otimizado pela primeira fase.

Mais informações sobre esta segunda abordagem podem ser encontradas em livros de compiladores, dos quais destacamos [13] e [31].

11.2.3 Terceira Abordagem

Um inconveniente da segunda abordagem é que a estrutura intermediária deve ser reconstruída a cada nova execução. Dependendo do tamanho do código e da presença de bibliotecas, isto pode tornar-se caro computacionalmente. Uma solução popular é salvar a estrutura intermediária em arquivo após a primeira fase. Neste caso, a primeira fase é um

⁵ *Abstract Syntax Tree - AST*

compilador que ao invés de gerar uma estrutura de dados intermediária, gera código executável para uma arquitetura própria (normalmente um ambiente virtual de execução ou máquina virtual).

Esta é a abordagem adotada por Java. O código fonte é uma linguagem de programação (chamada Java), que passa por um compilador e gera código executável que pode ser salvo em arquivo (no caso de java, um arquivo com extensão `.class`).

Assim como nas máquinas reais, as instruções do código executável são divididas em código de operação e operandos. Normalmente o código de operação é armazenado em um byte, o que explica o porquê destas instruções serem chamadas *bytecode*.

O arquivo executável pode ser lido por um interpretador e executado em uma máquina virtual própria para executar programas orientados a objeto, chamada Máquina Virtual Java, JVM⁶.

É interessante observar que as instruções *bytecode* Java são complexas, incluindo instruções para criar objetos (`new`), vetores (`newarray`) e matrizes multidimensionais (`multianewarray`). Para mais informações sobre *bytecode* e a JVM consulte [47].

Para uma abordagem prática para construir um compilador que gera código para a JVM veja [27].

11.2.4 Observações

Uma das vantagens do uso de linguagens interpretadas está no fato delas serem multiplataforma. Dado um programa escrito em uma destas linguagens, só é necessário ter o programa interpretador executável apropriado para que ele funcione naquela arquitetura.

Por outro lado, um programa executável é sempre mais rápido que seu equivalente interpretado (para o mesmo código). Porém, considerando o aumento da velocidade dos processadores modernos, a diferença pode não ser sentida pelos usuários.

Exercícios

- 11.1 Um programa que reproduz o comportamento de uma CPU é um simulador, emulador ou interpretador?
- 11.2 O programa `gdb` é um emulador, interpretador ou simulador?
- 11.3 Considere um interpretador em tempo de execução. Após ele ler o programa a ser interpretado, o que contém as seções `.text`, `.data`, `.bss` e na pilha? Onde estão armazenadas as variáveis do programa interpretado? Proponha uma forma de implementar chamadas de procedimento e alocação dinâmica para o programa interpretado.
- 11.4 Navegadores de internet contêm interpretadores para HTML e Javascript. Considere a possibilidade de compilar o código HTML e Javascript para um tipo de *bytecode*. Cite vantagens e desvantagens desta abordagem.
- 11.5 Webassembly é uma tecnologia que permite que o programador escreva um programa em, por exemplo, C, e gere *bytecode* Webassembly, que por sua vez é transmitido ao navegador que o interpreta num interpretador escrito em Javascript. Como você classifica esta abordagem?
- 11.6 Grande parte das aplicações Web são desenvolvidas em linguagens interpretadas (Ruby, Perl, PHP, .NET, etc.). Explique por que.
- 11.7 Por que as linguagens orientadas a objeto modernas são interpretadas? Para responder, é necessário entender os conceitos de orientação a objetos e analisar que tipo de funcionalidade é difícil de implementar em arquivos executáveis (por exemplo, ELF) que podem ser implementados no grafo de execução de linguagens interpretadas.
- 11.8 Considere um programa que contém um erro sintático, por exemplo, falta um ponto e vírgula ao final de um comando. Ao ser executado pelo interpretador, o programa só acusa erro se o fluxo de execução passar por aquele comando. Pergunta-se:
 - Qual a abordagem utilizada pelo interpretador?
 - Quais cuidados os programadores devem tomar ao testar este programa antes de colocá-lo em produção?
- 11.9 O termo utilizado para o interpretador Java foi “interpretador”, mesmo parecendo que ele parece mais um simulador. Explique por que ele não é um simulador. Para isto pense no que ele faz além de simular a execução dos *bytecodes*.

⁶ Java Virtual Machine

APÊNDICES

A. Arquiteturas de Fluxo de Controle

Este apêndice descreve o funcionamento de computadores baseados na arquitetura von Neumann (também conhecido como arquitetura de fluxo de controle¹), ou fluxo de instruções.

Esta categoria engloba a esmagadora maioria dos computadores comerciais, e pode ser resumida na Figura 34.

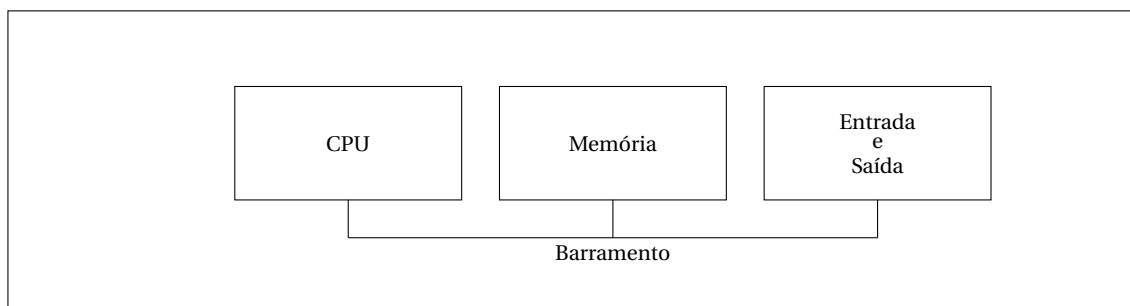


Figura 34 – Arquitetura de computadores baseados em fluxo de controle (arquitetura Von Neumann)

Esta figura mostra três componentes ligados por um barramento. A CPU é o componente que executa programas armazenados na memória. A execução ocorre com a repetição dos seguintes passos:

1. A CPU envia um endereço para a memória através do barramento. O endereço enviado é onde se espera estar a próxima instrução a ser executada.
2. A memória envia o conteúdo do endereço solicitado também através do barramento de volta para a CPU;
3. A CPU executa a instrução enviada;
4. A CPU calcula qual o endereço da próxima instrução.

Grande parte desta arquitetura baseia-se no trabalho de John von Neumann *et alii* [49]².

Baseado no trabalho de von Neumann *et alii*, foi desenvolvido o computador EDSAC que tornou-se operacional em 1949. Este computador tinha dois registradores e sua memória continha 256 palavras de 36 bits[26].

De lá para cá, outros modelos foram propostos como por exemplo, a arquitetura baseada em fluxo de dados (*dataflow architecture*[48]). A arquitetura baseada em fluxo de instruções sofreu poucas alterações conceituais, e a esmagadora maioria dos processadores comercialmente disponíveis seguem este modelo. Uma mudança grande está no crescimento vertiginoso de registradores e de memória disponível.

Para exemplificar, considere o intel 8080 (de 1974) que tinha sete registradores de oito bits sendo capaz de endereçar até 2^{16} (65K) bytes. O AMD64 utilizado neste livro, que pode ser visto como um “descendente” do intel 8080, tem 16 registradores de 64 bits de propósito geral, 8 registradores de 64 bits para ponto flutuante, 16 registradores de 128 bits, mais alguns de propósito específico, e tem capacidade para endereçar até 2^{64} bytes.

Este capítulo descreve os principais componentes de armazenamento do AMD64 (registradores e memória), assim como um subconjunto de instruções.

Para tal, o restante deste apêndice foi dividido para detalhar os principais aspectos: a Seção A.1 descreve os registradores, a Seção A.2 descreve como funciona a memória e a Seção A.3 descreve algumas das instruções.

A.1 Registradores

Os registradores do AMD64 usados neste livro são todos de 64 bits. Para permitir que o processador fosse usado também por programas projetados para máquinas de 32 bits (chamados de x86 – 32), estes registradores coexistem com os de 32

¹ Control flow architecture

² em conjunto com os engenheiros Eckert e Mauchly que injustamente não receberam igual crédito[16]

bits. Há uma diferença no nome deles: os de 32 bits são precedidos pela letra *e* enquanto que os de 64 bits são precedidos pela letra *r*. Assim, *%eax* é um registrador de 32 bits enquanto que *%rax* é um registrador de 64 bits.

É comum que cada registrador seja nomeado individualmente de duas formas: pelo seu número ou pelo seu apelido. A Tabela 21 lista os registradores de propósito geral do AMD64 usados neste livro. É interessante observar que os registradores *%r8–%r15* não tem nomes, só números.

Número	Nome	Número	Nome
<i>%r0</i>	<i>%rax</i>	<i>%r8</i>	<i>%r8</i>
<i>%r1</i>	<i>%rcx</i>	<i>%r10</i>	<i>%r10</i>
<i>%r2</i>	<i>%rdx</i>	<i>%r11</i>	<i>%r11</i>
<i>%r3</i>	<i>%rbx</i>	<i>%r9</i>	<i>%r9</i>
<i>%r4</i>	<i>%rsp</i>	<i>%r12</i>	<i>%r12</i>
<i>%r5</i>	<i>%rbp</i>	<i>%r13</i>	<i>%r13</i>
<i>%r6</i>	<i>%rsi</i>	<i>%r14</i>	<i>%r14</i>
<i>%r7</i>	<i>%rdi</i>	<i>%r15</i>	<i>%r15</i>

Tabela 21 – Relacionamento entre os números e nomes dos registradores

Vários assuntos foram omitidos aqui por fugirem ao escopo do livro, como por exemplo, onde residem os registradores para compatibilidade (de 8, 16 e 32 bits). Isto pode ser encontrado em [46].

Um registrador importante, mas que não está listado acima é o *%rip* (*instruction pointer*). Ele armazena sempre o endereço da próxima instrução a ser executada. Todas as arquiteturas baseadas na arquitetura von Neumann tem um registrador equivalente. Na maioria delas, este registrador é chamado “program counter”, ou PC, mas seguindo a nomenclatura dos processadores da linha x86, eles são chamados *instruction pointer*, o que faz mais sentido que *program counter*, uma vez que ele aponta para o endereço da próxima instrução.

A.2 Memória

A maneira mais simples de descrever o que é e como acessar a memória é comparando-a a um vetor de n posições. Para acessar o conteúdo da posição x ($0 \leq x \leq n - 1$), faz-se algo como $M[x]$. Porém a forma de executar esta ação pode ser melhor detalhada:

1. a CPU coloca o endereço da posição de memória a ser acessada no barramento (no exemplo, x);
2. a memória recebe o endereço e coloca no barramento o conteúdo daquela posição de memória ($M[x]$);
3. a CPU armazena $M[x]$, que contém a instrução a ser executada.

A memória armazena também instruções. Logo, é possível executar o procedimento acima para buscar a próxima instrução da memória. Com isto explicado, podemos ser mais precisos sobre como a CPU do AMD64 faz para executar programas, que é basicamente repetindo os seguintes passos:

1. A CPU coloca o valor de *%rip* no barramento.
2. A memória envia o conteúdo do endereço indicado através do barramento de volta para a CPU;
3. A CPU decodifica e executa a instrução enviada;
4. A CPU atualiza o *%rip* para o endereço da próxima instrução.

A.3 Conjunto de Instruções

A seção anterior explicou como se busca instruções, mas não o que são as instruções, nem sobre quais componentes ela trabalha. Isto será explicado nesta seção.

Cada CPU tem um conjunto específico de instruções. Estas instruções são as responsáveis por transportar valores entre os componentes do computador (em especial entre memória e registradores), executando também operações sobre os valores contidos nos componentes (operações aritméticas e lógicas).

Podemos dividir uma instrução em duas partes: (1) o que ela faz e (2) sobre que componentes ela opera. Por exemplo, a instrução que copia o valor contido em um registrador para outro registrador se divide em (1) copiar e (2)

especificar os registradores origem e destino. Na taxonomia de arquitetura de computadores, estas partes são chamadas (1) “operação” e (2) “operandos”.

O significado da operação é algo até intuitivo, como por exemplo copiar, somar, subtrair, etc. Os operandos indicam sobre quais “objetos” executar a operação. Estes “objetos” podem ser ou constantes ou registradores ou memória.

As instruções de todas as CPUs (que eu conheço) estão divididas nestas duas partes. Algumas operações precisam de somente um operando, outras precisam de dois ou mais operandos. Curiosamente, existem operações que em uma CPU precisam de um certo número de operandos e em outra CPU de um número diferente de operandos.

Como exemplo, considere a operação soma. Ela atua sobre três objetos (no mínimo): um destino e duas origens. Se considerarmos que a instrução é soma *operando1*, *operando2*, *operando3*, a ação é

$$\text{operando1} \leftarrow \text{operando2} + \text{operando3}.$$

É natural pensar que em todas as CPUs, esta instrução tenha um código de operação (soma) e três operandos. Porém, existem CPUs (como o AMD64) que exigem somente dois operandos, onde um operando atua como origem e como destino:

$$\text{operando1} \leftarrow \text{operando1} + \text{operando2}$$

Vamos agora nos aprofundar na questão dos operandos.

Como já dito anteriormente, os operandos indicam quais os objetos devem ser usados na operação indicada. Estes objetos são: constantes, registradores e memória.

Os “modos de endereçamento” são a formalização de quais objetos são referenciados. Ao longo do tempo, vários modos de endereçamento foram implementados em diferentes CPUs (veja [16] para mais detalhes).

No AMD64 foi implementado um conjunto menor de modos de endereçamento, e os exemplos deste livro concentram-se em um conjunto ainda menor (somente em cinco deles): endereçamento registrador, endereçamento imediato, endereçamento direto, endereçamento indireto e endereçamento indexado³.

Nas próximas seções, serão descritos estes modos de operação e como eles são utilizados no AMD64. Em todos os casos, será utilizada como exemplo a instrução

```
movq operando1, operando2
```

, que copia⁴ o conteúdo do *operando1* para o *operando2*. Observe o sufixo *q*. As instruções que operam sobre registradores de 32 bits utilizam o sufixo *l* (como por exemplo, *movl %eax, %ebx*) enquanto que as instruções que operam sobre instruções de 64 bits utilizam o sufixo *q* (*movq %rax, %rbx*)⁵.

A.3.1 Endereçamento Registrador

É um modo de endereçamento onde os operandos são registradores. O exemplo abaixo copia o conteúdo do registrador *%rax* para *%rbx*.

```
movq %rax, %rbx # %rbx ← %rax
```

Neste caso, tanto o primeiro quanto o segundo operandos utilizam o endereçamento registrador.

A.3.2 Endereçamento Imediato

São instruções que lidam com constantes. Uma constante é indicada com o símbolo “\$” à frente de um número (ou rótulo). Também é possível indicar constantes em hexadecimal acrescentando-se um “0x” à frente do número, como por exemplo, “\$0x80”.

```
movq $79, %rax # %rax ← 79
movq $A, %rax # %rax ← A
```

A primeira linha do exemplo acima copia a constante 79 para o registrador *%rax*.

Na segunda linha, o símbolo *A* é rótulo, ou seja, um mnemônico para indicar um endereço de memória. Este endereço e não o conteúdo do endereço (ou seja, *x* e não *M[x]*) é copiado para dentro do registrador *%rax*.

³ taxonomia de [16]

⁴ uma questão semântica cujo motivo foi perdido na história. Apesar da instrução ser *mov*, abreviatura de *move* (mover), a ação que ele executa é de cópia

⁵ *q* significa “quadword” enquanto que *l* significa “long”

A.3.3 Endereçamento Direto

Endereçamento direto é um dos três modos de indicar a memória como operando.

São instruções que copiam constantes ou valores contidos em registradores em endereços específicos de memória.

Exemplo:

$$M[\text{<constante>}] \leftarrow 79$$

O valor 79 será copiado para dentro da posição de memória indicada por <constante>.

Exemplos de endereçamento direto no AMD64:

```
movq    0x79,    %rax    # %rax ← $79
movq    A,       %rax    # %rax ← M[$A]
```

No primeiro caso, o conteúdo da posição 0x79 (hexadecimal) é copiado para %rax, enquanto o segundo caso é a operação inversa.

No terceiro caso, o símbolo A é um rótulo, e o conteúdo do endereço daquele rótulo é copiado para %rax.

Observe que precedendo o operando do endereçamento direto com um \$, o modo de endereçamento muda para imediato.

A.3.4 Endereçamento Indireto

São instruções que copiam valores para dentro de endereços de memória indicados por registradores.

$$M[\%rax] \leftarrow 12$$

O valor 12 será copiado para a posição de memória indicada pelo registrador %rax.

A forma de indicar endereçamento indireto é colocando parênteses em volta do nome do registrador. Exemplos de instrução com endereçamento indireto no AMD64:

```
movq    $0x71,    (%rax)    # M[%rax] ← $0x71
movq    (%rax),   %rax      # %rax ← M[%rax]
```

O endereço de memória acessado é o endereço contido em %rax.

```
movq    $0x73,    A(%rax)    # M[%rax + $A] ← $0x73
movq    A(%rax),  %rax      # %rax ← M[%rax + $A]
```

O endereço de memória acessado é a soma do rótulo A (uma constante) com o endereço contido em %rax.

```
movq    $0x75,    -8(%rax)    # M[%rax + (-8)] ← $0x75
movq    -8(%rax), %rax      # %rax ← M[%rax + (-8)]
```

O endereço de memória acessado é obtido somando-se o conteúdo de %rax com -8.

Estes dois últimos exemplos são muito utilizados para acessar variáveis locais e parâmetros em um registro de ativação.

A.3.5 Endereçamento Indexado

Este é o último tipo de endereçamento utilizado neste livro, e foi projetado para facilitar o acesso a vetores. Este endereçamento utiliza três parâmetros para acessar a memória: endereço base, deslocamento e o tamanho.

$$M[\text{Base} + \text{Desloc} * \text{tamanho}] \leftarrow 12$$

O cálculo acima soma o endereço base do vetor (um rótulo) com o deslocamento (o registrador %rdi) multiplicado pelo tamanho de cada elemento do vetor. Se for um inteiro (que ocupa oito bytes), o tamanho é 8, se for um caractere o tamanho é 1 byte e assim por diante.

Exemplo no AMD64:

```
movq    $0x81,    A(,%rdi,8)
```

Este endereçamento é útil para acessar vetores, uma vez que é necessário colocar no registrador `%rdi` o mesmo valor que seria colocado no subscrito do vetor.

Por exemplo, considere a instrução em C abaixo:

```
a[i]=129
```

Suponha que o valor da variável `i` está contido em `%rdi`, e que o endereço base do vetor é indicado pelo rótulo `A`. Neste caso, a instrução equivalente em assembly é:

```
movq    $129,    A(,%rdi,8)
```

A.4 Formato das Instruções

Um aspecto que será repetidamente analisado ao longo do livro está relacionado com a forma com que as instruções são representadas em arquivo e na memória em tempo de execução.

Quando uma instrução é trazida da memória para a CPU, o formato não é o mesmo que o mnemônico dela. Como exemplo, considere a instrução

```
movq    %eax,    %ebx
```

O formato acima é a representação da instrução em caracteres. Acredito não surpreender ninguém ao explicar que os caracteres acima não são trazidos da memória para a CPU executar. O que é trazido é uma versão compacta desta instrução, chamado formato de máquina ou binário.

Normalmente uma instrução codificada em binário contém quatro informações: código de operação (qual a operação a ser realizada), e os operadores onde serão realizadas: operador origem1, operador origem2 e operador destino.

Cada um destes espaços codificados ocupam um certo número de bits na instrução. Em máquinas RISC, este formato é “ortogonal”, ou seja cada pedaço, ou campo, ocupa sempre os mesmos bits da instrução.

Como exemplo, considere uma instrução de 32 bits onde os bits 26-31 indicam o código de operação (seis bits, permitindo 64 instruções diferentes), e cada operando ocupa 5 bits (indicando 32 registradores)⁶.

Infelizmente, as instruções do AMD64 não são ortogonais, ou seja, as instruções têm tamanhos diferentes (variando de um a quinze bytes[45]), onde cada campo utiliza número diferente de bits.

Além disto, existe um campo adicional, o `prefixo`, que é um byte que modifica a ação da instrução, ou operandos.

Descrever o formato exato das instruções é uma tarefa extremamente complexa que foge ao escopo deste livro. Para dar uma ideia desta complexidade, veja a Figura 35.

O que faremos é mostrar como é a codificação de algumas poucas instruções, permitindo ler um pouco melhor um arquivo binário objeto ou executável ou ainda um programa em execução.

Para tal, considere a instrução abaixo:

```
movq    %rax,    %rbx
```

Cuja representação em hexadecimal é `0x4889c3`. Esta instrução está dividida em quatro partes: prefixo, código de operação, operando 1 e operando 2, mas só é possível distingui-los na sua representação em binário:

```

01001000 1000100111 000 011
  prefixo   Cod.Oper.  Op1  Op2

```

prefixo É uma espécie de modificador da instrução;

código de operação indica a ação a ser efetuada (no caso, MOV);

operando 1 indica registrador destino (`%rax`);

operando 2 indica registrador origem (`%rbx`).

Uma incoerência pode ser observada: temos 16 registradores de 64 bits, e somente três bits para indicá-los (ou seja, somente oito registradores). Somente os registradores zero até sete podem ser endereçados assim. Para endereçar os demais, há um truque: modificar o prefixo. É algo inusitado, que não parece ser adequado, mas é exatamente o que é feito.

⁶ O processador MIPS codifica suas instruções desta forma

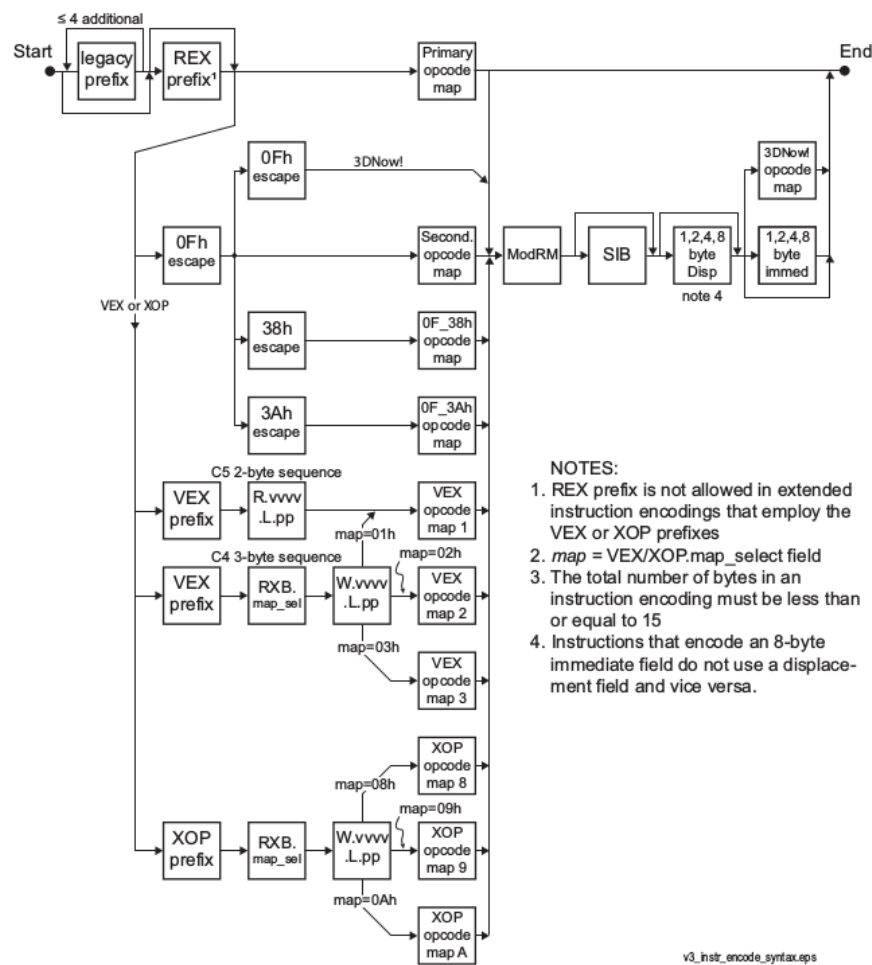


Figure 1-1. Instruction Encoding Syntax

Figura 35 – Sintaxe da codificação de uma instrução (fonte:[46])

Quando o prefixo for 0x48, os registradores afetados são os que têm o bit 4 igual a zero, ou seja %r0 até %r7 (primeira coluna da Tabela 21). Quando o prefixo for 0x49, os registradores afetados são os que têm o bit 4 igual a 1, ou seja %r8 até %r15 (segunda coluna da Tabela 21)⁷.

Este tipo de “artifício” é comum nos processadores derivados do intel 8080. É provável que após esta observação, fique mais claro porque os professores da disciplina de arquitetura de computadores têm alguma resistência a adotar esta família como alvo de estudo.

⁷ Em português, temos várias palavras para descrever isto. Elas vão de maracutaia, chuncho até o mais politicamente correto “artifício”

B. Formatos de Instrução

Este apêndice aborda alguns dos formatos de instrução nos processadores da família AMD-64¹. Com formato de instrução, entende-se a forma com que a instrução é organizada fisicamente para conter código de operação e os operandos (se houverem).

Este tópico é mais simples quando analisamos processadores RISC (reduced instruction set computer), onde código de operação e operandos estão organizados de maneira regular em dois ou três formatos diferentes. Nestas máquinas, o código de operação tem um tamanho fixo, e ocupa o mesmo espaço em qualquer dos formatos. Já em processadores CISC (complex instruction set computer), não há esta regularidade, e os códigos de operação podem ocupar de poucos a muitos bits e vários formatos de instrução são adotados, o que torna difícil determinar quando termina o código de operação e quando começam os operandos.

Este é um dos aspectos que fazem com que os processadores RISC sejam considerados “melhores” do que os computadores CISC (apesar de isso nem sempre ser verdade). Nas máquinas CISC, como as instruções aparecem em grande número e são complexas, a unidade de controle deve ser maior para contemplar todas as regras e todas as exceções dos vários formatos. A CPU deve ficar maior, e por consequência as distâncias entre os componentes internos ficam maiores também, fazendo com que os dados demorem mais para trafegar entre os componentes (por exemplo, entre registradores). O efeito da implementação “direta” de um processador CISC é que as suas instruções ficam mais lentas.

Por outro lado, a regularidade das instruções RISC faz com que a unidade de controle seja mais simples, e consequentemente menor. Com isso, a CPU pode ser projetada de forma mais simples e as distâncias entre os componentes também ficam menores. A consequência é que as instruções demoram menos tempo para serem completadas, e o efeito é que elas passam a ser mais rápidas.

Os processadores AMD64 são exemplos de processadores CISC e honram a classe. Eles contêm centenas de instruções cujo tamanho varia de um a quinze bytes.

Aqui o leitor deve estar se perguntando: então porque ainda são produzidos computadores com processadores CISC ao invés de adotar processadores RISC? A resposta está no preço e na compatibilidade com processadores anteriores. Por exemplo, cada novo processador da família x86 (inclua-se aí os x86-64) executa o conjunto de instruções de seus predecessores de forma mais rápida que gerações anteriores (maravilhas da engenharia), acrescentam algumas instruções novas de acordo com as necessidades de mercado (como por exemplo, MMX e SSE) e com isso tem mercado praticamente garantido para venda. Com a massificação, os custos de projeto e de produção são minimizados (para não dizer pulverizados). Já os processadores RISC nem sempre tem mercado garantido (pelo menos não na magnitude dos x86). Com isso, tanto os custos de projeto quanto os de produção podem ser sentidos no preço final do processador.

B.1 Modos de Endereçamento

Os operandos de uma instrução em assembly podem variar de acordo com o local em que o dado se encontra. Como exemplo, observe a diferença entre as instruções:

- `movq %rax, %rbx` (endereçamento registrador)
- `movq $0, %rbx` (endereçamento imediato)
- `movq A, %rbx` (endereçamento direto)
- `movq %rax, (%rbx)` (endereçamento indireto)
- `movq A(%rdi,4), %rbx`. (endereçamento indexado)

Estes são os modos de endereçamento que esta seção detalhará. Existem outros modos de endereçamento que não os descritos acima mas como eles não foram implementados nesta família de processadores, não serão abordados aqui como por exemplo, onde os dois operandos estão em memória (algo em desuso nos processadores modernos).

Antes de iniciar a explicação, existem dois aspectos que queremos destacar:

¹ também conhecidos como x86-64, x64, x86_64

1. Nos exemplos acima foi usada somente a instrução `movq`. Apesar de ser uma única instrução, ela foi projetada para conter um número variado de operandos. Cada um destes modos de endereçamento da instrução `movq` tem um código de operação (ou modificador) diferente, o que os diferencia internamente para o processador.
2. A nomenclatura dos modos de endereçamento (Registrador, Imediato, Direto, Indireto e Indexado) não é única. Os manuais do AMD64 usam uma taxonomia um pouco diferente: imediato, registrador, memória (que aqui está dividido em base/deslocamento e base/índice/deslocamento) e portas de I/O (que não será coberto aqui). Para maiores detalhes veja [46].

B.2 Registrador

Este modo de endereçamento envolve dois operandos que são registradores.

Como exemplo, considere a instrução

```
movq %rax, %rbx,
```

onde o conteúdo do registrador `%rax` será copiado² para `%rbx`. A instrução contém três informações: código de operação, operando 1 (`%rax`) operando 2 (`%rbx`). O modo de endereçamento está implícito no código de operação (ou seja, o código de operação neste caso é: copie o valor de `%rax` para `%rbx` usando endereçamento registrador). Para outros modos de endereçamento, o código de operação é diferente.

Os detalhes de endereçamento da instrução `movq` podem ser encontrados em [45], no comando `mov`.

Cuja representação em hexadecimal é `0x4889c3` (verifique no código binário executável). Esta instrução está dividida em quatro partes: prefixo, código de operação, operando 1 e operando 2, mas só é possível distingui-los em binário:

0100	1000	1000	1001	11	000	011
prefixo		Cod. Oper.		Op1	Op2	

Nestes bits estão armazenados em quatro “campos”:

prefixo É uma espécie de modificador da instrução;

código de operação indica a ação a ser efetuada (no caso, MOV);

operando 1 indica registrador destino (`%rax`);

operando 2 indica registrador origem (`%rbx`).

Nesta instrução particular, o modificador ocupa oito bits, o código de operação ocupa 10 bits, o operando destino ocupa 3 bits e o operando origem também ocupa três bits.

Observe que o registrador `%rax` é $(000)_2 = 0_{10}$, enquanto que o registrador `%rbx` é $(011)_2 = 3_{10}$. Seria natural esperar que o registrador `%rbx` fosse o de número 1, porém a relação é diferente, como pode ser visto na Tabela 22.

Como estes são os 16 registradores de propósito geral, seus números podem ser representados em quatro bits. Como os operandos só podem representar três bits, o bit mais à esquerda vem de outro lugar, no caso do prefixo. Para indicar isto, a notação da tabela mostra que o registrador `%rax` é representado como `0 000` ao invés de `0000` como seria natural.

O motivo desta notação é deixar claro que os três bits à direita são indicados no operando, enquanto que o bit à esquerda é indicado no campo prefixo (ou modificador): Para indicar “0”, o prefixo é `0x48` (como no exemplo) e para indicar “1”, o prefixo é `0x49`.

Este é um “artifício” esquisito, mas estas “esquisitices” são comuns em processadores CISC, o que os tornam um alvo não muito próprio para ensinar arquitetura de computadores.

² Observe que apesar da instrução ser “move” (mover em português), a operação é de cópia.

Registrador	Número
rax	0 000
rcx	0 001
rdx	0 010
rbx	0 011
rsp	0 100
rbp	0 101
rsi	0 110
rdi	0 111
r8	1 000
r10	1 001
r11	1 010
r9	1 011
r12	1 100
r13	1 101
r14	1 110
r15	1 111

Tabela 22 – Números internos associados aos registradores AMD64

B.3 Imediato

Este é o modo de endereçamento que envolve uma constante (também chamada de valor imediato) que está incluída na própria instrução. Nos AMD64, é possível utilizar constantes de vários tamanhos (8, 16, 32 e 64 bits) mudando o código de operação.

Como exemplo, considere a instrução `movq $1, %rax`, cuja representação em hexadecimal é

```
48 c7 c0 01 00 00 00
```

Em binário, temos:

```

0100 1000 1100 0111 1100 0 000 0000 0001 0000 0000 0000 0000 0000
prefixo      CO      OP1      Imediato

```

O prefixo ocupa os primeiros oito bits. Em seguida, o código de operação com 13 bits³. Em seguida, os três bits do operando 1 (`%rax`) e finalmente a constante que ocupa os últimos 32 bits. Lembre que os AMD64 são *little endian* (veja Seção C.2) e como a memória é crescente os bytes aparecem invertidos.

Como exercício, veja a representação da instrução

```
movq $0x123456789abcdef
```

Observe que diferente do outro exemplo, o operando imediato contém 48 bits (ao invés de 32). Aproveite e verifique como fica uma instrução com um imediato de 64 bits.

B.4 Endereçamento Direto

No endereçamento direto, a instrução contém o endereço do dado a ser utilizado, como por exemplo, na instrução `movq B, %rbx`, onde “B” é um rótulo na seção data (lembre-se que os endereços dos rótulos são definidos em tempo de ligação). Assumindo que o endereço do rótulo “B” é `0x600105`, então a instrução será codificada como `48 8b 1c 25 05 01 60 00`, ou em binário:

```

0100 1000 1000 1011 1100 0 011 0010 0101 0000 0101 0000 0001 0110 0000 0000 0000
prefixo      CO      OP1      Imediato

```

³ a realidade é menos elegante (se é que isso é possível), mas optamos por uma interpretação mais simples pois uma explicação completa da organização do código de operação do AMD64 foge ao escopo do livro (além de ser bastante confusa).

A instrução “inversa” é `movq %rbx, %rax`, cujo código binário é `48 89 1c 25 05 01 60 00`. É interessante observar que a única mudança está no segundo byte (código de operação), e que os parâmetros podem ser encontrados nos mesmos locais.

B.5 Endereçamento Indireto

No modo de endereçamento indireto, um dos parâmetros indica uma referência indireta à memória (normalmente um registrador cercado de parênteses).

Por exemplo, a instrução `movq (%rbx), %rax` copia o conteúdo do endereço contido no registrador `%rbx` para o registrador `%rax`, algo como:

$$\%rax \leftarrow M[\%rbx]$$

Onde $M[\%rbx]$ indica o conteúdo de memória indexado por `%rbx`.

A instrução `movq (%rbx), %rax` é codificada em hexadecimal como `48 8b 03`, e em binário, temos:

$$\begin{array}{ccccccc} 0100 & 1000 & 1000 & 1011 & 0 & 000 & 0 & 011 \\ \hline \text{prefixo} & & CO & & OP1 & \text{Sem uso} & OP2 \end{array}$$

Observe que há um bit sem uso fixado em zero que não é usado (coisa comum em máquinas CISC).

Lembre-se que o prefixo é usado para indicar o uso dos “primeiros” ou “últimos” oito registradores. Aqui ele também é usado para indicar se `OP1` ou `OP2` é o operando utilizando endereçamento indireto. Veja com `movq %rbx, (%rax)`.

AVISOS IMPORTANTES

- “Dupla indireção”, ou seja, `((%reg))` não é válido.
- Somente um operando pode ser indireto. Por isso, `movq (%rbx), (%rax)` não é válido.
- É possível usar endereçamento indireto com rótulos, ou seja, são válidas as instruções do tipo `movq (A), %rax`.

B.6 Endereçamento Indexado

O que aqui chamamos de endereçamento indexado também é conhecido como endereçamento de memória base e deslocamento.

No endereçamento indexado, a instrução usa um “endereço base” e um deslocamento. Um exemplo é a instrução `movq A(,%rdi, 8), %rbx`, que usa “A” como base e `%rdi × 8` como deslocamento. A instrução pode ser melhor entendida pela fórmula

$$\%rbx \leftarrow M[\&A + \%rdi \times 8]$$

onde `&A` indica o endereço de “A”.

Uma outra forma de ver o funcionamento deste tipo de endereçamento é descrito no Algoritmo B.1.

1	<code>movq \$A, %rax</code>
2	<code>mulq 8,%rdi</code>
3	<code>addq %rdi, %rax</code>
4	<code>movq (%rax), %rbx</code>

Algoritmo B.1 – Ação equivalente à da instrução `movq A(,%rdi, 8), %rbx`

Observe que o segundo parâmetro (o registrador `%rdi`) é multiplicado pelo terceiro parâmetro. Isto é usado quando os elementos do vetor estão distantes um número constante de bytes (por exemplo, para inteiros, esta constante é 8 bytes). Se fosse um vetor de caracteres, a instrução seria `movq A(,%rdi, 1), %rbx`. Ou seja, o terceiro parâmetro indica o tamanho do tipo de dado contido no vetor.

Agora, vamos analisar o formato desta instrução. Aqui, se assumirmos que o endereço de “A” é `0x6001cb`, obteremos a seguinte instrução em hexa: `48 8b 1c fd 8b 01 60 00`. Ao analisar o código binário, temos:

0100 1000 1000 1011 0	0	011	0	111	...
$\underbrace{\hspace{1.5cm}}$	$\underbrace{\hspace{0.5cm}}$	$\underbrace{\hspace{0.5cm}}$	$\underbrace{\hspace{0.5cm}}$	$\underbrace{\hspace{0.5cm}}$	$\underbrace{\hspace{0.5cm}}$
<i>prefixo</i>	<i>CO</i>	<i>sem uso %rbx</i>	<i>sem uso %rdi</i>	<i>end.A</i>	

Como já sabemos, o prefixo indica se o primeiro e o segundo parâmetros são os oito primeiros ou últimos registradores.

Depois, o código de operação, um bit sem uso e o registrador destino. Novo bit sem uso e o registrador origem.

Por fim, o endereço de A.

B.7 Conclusão

O processador AMD64 é um processador CISC. Por esta razão não surpreende que existam muitas instruções e vários modos de endereçamento. Também não surpreende que as instruções sejam de tamanhos diferentes (de 1 byte até 15 bytes).

Neste apêndice foram apresentados os modos de endereçamento adotados no livro, com instruções que operam sobre números inteiros, a CPU “normal”.

O AMD64 tem outros aspectos interessantes, como por exemplo, a presença de três outros conjuntos de registradores e de instruções que as manipulam.

SIMD Para atender a demanda por desempenho dos aplicativos que usam *streaming*⁴, foram desenvolvidas instruções que operam sobre múltiplos dados simultaneamente (ou seja SIMD⁵). Estas instruções utilizam 16 registradores de 256 bits.

MMX e 3DNow! Uma extensão ao modelo Streaming SIMD Extensions (SSE), que operam sobre os registradores MMX de 64 bits.

Ponto Flutuante para operações em ponto flutuante (números reais) que operam no padrão IEEE 754 and 854.

Com isto, é possível concluir que o AMD64 é composto por mais de uma CPU na mesma pastilha, e consequentemente, a complexidade embutida neste processador é notável.

Para fugir desta complexidade, este livro aborda somente uma pequena parte do AMD64, aquela que lida com operadores inteiros.

Com estes comentários, fica bastante simples de entender porque esta CPU raramente é usada na academia para estudo. Exceção feita a professores sádicos, claro.

⁴ arquivos grandes lidos em sequência, normalmente usados para multimídia (áudio, vídeo)

⁵ Single Instruction Strem Multiple Data Stream

C. Depurador

Um depurador é um aplicativo que simula a execução de um programa em uma dada arquitetura. No caso deste livro, isto é particularmente útil para analisar o que ocorre com o programa em tempo de execução.

Este apêndice descreve o funcionamento de um destes depuradores, o `gdb` (Gnu Debugger). Qualquer outro depurador pode ser utilizado, como por exemplo, o `ddd` (Data Display Debugger) que tem a vantagem de ser gráfico enquanto o `gdb` só opera em modo texto. O importante é que o depurador escolhido seja capaz de:

1. executar as instruções *assembly* dos programas passo a passo (uma instrução *assembly* por vez);
2. criar pontos de parada (*breakpoints*);
3. mostrar a representação binária das instruções;
4. listar o valor de cada registrador ao longo da execução.
5. mostrar a memória;

C.1 GDB

Para executar o `gdb`, deve-se primeiro instalá-lo no sistema (até onde sei, ele não é instalado por padrão em nenhuma distribuição linux).

Para executá-lo (ou o `ddd`), há um truque: eles só funcionam com um valor específico da variável de ambiente `SHELL`:

```
> export SHELL=/bin/sh
> gdb
```

Para exemplificar o funcionamento, iremos usar o programa `esqueletoS.s` reproduzido no Algoritmo C.1:

```
1 .section .data
2 .section .text
3 .globl _start
4 _start:
5     movq $60, %rax           # %rax := 60
6     movq $13, %rdi          # %rdi := 13
7     syscall
```

Algoritmo C.1 – Reimpressão do arquivo `esqueletoS.s`

Para executá-lo, é necessário primeiro gerar um executável com os comandos abaixo. O executável aqui tem o nome `esqueletoS`.

```
> as esqueletoS.s -o esqueletoS.o -g
> ld esqueletoS.o -o esqueletoS -g
```

A opção “-g” inclui informações de depuração (gera formato `DWARF`). Por fim, para depurá-lo, iniciar o programa `gdb`:

```
> gdb
NU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
```

Type "show configuration" for configuration details.
 For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
 Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
 For help, type "help".
 Type "apropos word" to search for commands related to "word".
 (gdb)

A última linha (gdb) é onde o usuário pode digitar comandos. São muitos mas vamos usar bem poucos aqui. Listamos todos na abaixo:

Comando	Descrição
file <arq>	carrega o arquivo executável <arq>
break <rotulo>	coloca um ponto de parada
run	executa o arquivo indicado
step	executa uma única instrução assembly
info registers	mostra o valor de cada registrador
x/n addr	mostra o conteúdo de M[addr]

A sequência abaixo carrega o programa executável no simulador e coloca um ponto de parada antes de executá-lo. Isto é importante, pois o comando run executaria todo o programa sem permitir o passo a passo que desejamos.

```
...
(gdb) file esqueletoS
Reading symbols from esqueletoS...done.
(gdb) break _start
Breakpoint 1 at 0x400078: file esqueletoS.s, line 5.
(gdb) run
Starting program:
./esqueletoS
Breakpoint 1, _start () at esqueletoS.s:5
5      movq $60, %rax          # %rax := 60
(gdb) step
6      movq $13, %rdi         # %rdi := 13
(gdb) step
7      syscall
(gdb) step
[Inferior 1 (process 3632) exited with code 015]
```

A execução acima mostra pouca coisa. Seria mais interessante se fosse possível examinar o que ocorre com o valor dos registradores. Para isto, execute mais uma vez o programa, porém desta vez digite `info registers` antes e depois das instruções movq.

Outra coisa interessante seria ver como fica o programa na memória. Porém, como é compreensível, deve-se indicar qual a posição inicial e final.

Para isto, usa-se comando x. Ele tem um monte de parâmetros, conforme descrito abaixo:

```
...
(gdb) x/80x 0x400060
0x400060: 0x00000088 0x00000000 0x00000088 0x00000000
0x400070: 0x00200000 0x00000000 0x3cc0c748 0x48000000
0x400080 <_start+8>: 0x000dc7c7 0x050f0000 0x00000000 0x00000000
0x400090: 0x0000002c 0x00000002 0x00080000 0x00000000
0x4000a0: 0x00400078 0x00000000 0x00000010 0x00000000
0x4000b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x4000c0: 0x0000007d 0x00000002 0x01080000 0x00000000
0x4000d0: 0x00400078 0x00000000 0x00400088 0x00000000
0x4000e0: 0x75717365 0x74656c65 0x732e536f 0x6f682f00
0x4000f0: 0x622f656d 0x6c6c756d 0x542f7265 0x61626172
0x400100: 0x2f6f686c 0x63736944 0x696c7069 0x2f73616e
0x400110: 0x36304943 0x694c2f34 0x2f6f7276 0x7276694c
```

```

0x400120:      0x6e62416f      0x2f343674      0x502f4253      0x73616d67
0x400130:      0x554e4700      0x20534120      0x36322e32      0x0100312e
0x400140:      0x00110180      0x01110610      0x08030112      0x0825081b
0x400150:      0x00000513      0x00003c00      0x23000200      0x01000000
0x400160:      0x0d0efb01      0x01010100      0x00000001      0x01000001
0x400170:      0x71736500      0x656c6575      0x2e536f74      0x00000073
0x400180:      0x09000000      0x40007802      0x00000000      0x75751600
0x400190:      0x01000202      0x00000001      0x00000000      0x00000000

```

O comando `x/80x 0x400060` significa:

x imprime memória;

/80x oitenta palavras (quatro bytes) em hexadecimal;

0x400060 começando no endereço 0x400060.

O resultado é a lista de endereços e seus conteúdos. Por exemplo, as linhas abaixo relacionam endereços a valores. Porém, infelizmente há um truque aqui, que dificulta a compreensão exata do que está acontecendo. Este truque é chamado “Sequenciamento da informação”, que será visto na Seção C.2.

```

0x400070:      0x00200000      0x00000000      0x3cc0c748      0x48000000
0x400080 <_start+8>:  0x000dc7c7      0x050f0000      0x00000000      0x00000000

```

Esta informação permite entender como funciona a organização de processo na memória como mostrado na Figura 1 reproduzida na Figura 36.

Observe que a figura mostra os endereços com todos os 64 bits (16 símbolos hexadecimais) enquanto que o simulador não mostra os hexadecimais zero à esquerda. Em outras palavras, onde o simulador mostra 0x400070, a figura mostra 0x0000000000400070.

Também é importante destacar que o simulador indica a memória com endereços crescentes enquanto a figura mostra a memória com endereços decrescentes.

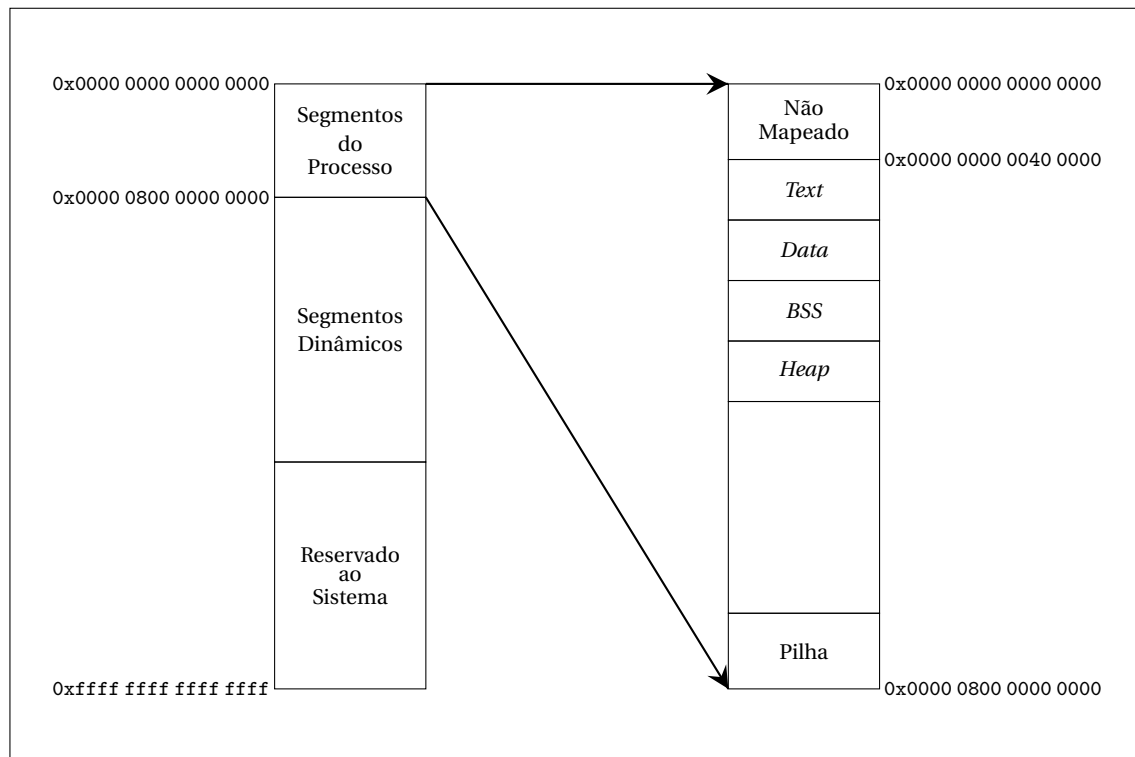


Figura 36 – Espaço Virtual de um Processo

Neste ponto é interessante analisar o simulador sob a perspectiva dos processadores baseados no fluxo de controle (Figura 34 reproduzida na Figura 37).

- O comando `info registers` permite ver o conteúdo dos registradores na CPU.
- O comando `step` traz o conteúdo de `M[%rip]` para a CPU, onde a instrução é decodificada.
- O comando `x` permite ver o conteúdo da memória a partir de endereços informados.
- O mapa da memória virtual de um processo (Figura 36) mostra como o processo é organizado na memória.
- Para ver o código hexadecimal da instrução, digite `set disassemble-next-line on`

Porém, (pelo menos) uma questão fica em aberto: no modelo apresentado, o computador funciona como se houvesse um único processo em memória. Já que sabemos que é possível que vários processos executem simultaneamente, em princípio todos deveriam usar a memória também.

Isto é obtido com o conceito de memória virtual, apresentado resumidamente no Apêndice D.

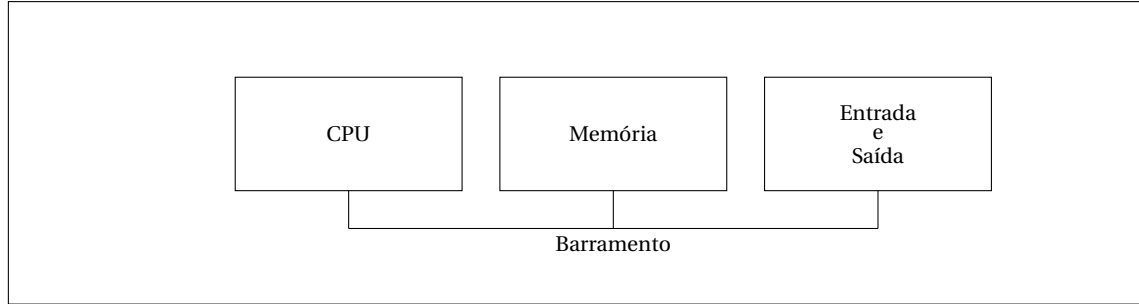


Figura 37 – Arquitetura de computadores baseados em fluxo de controle (arquitetura Von Neumann)

C.2 Sequenciamento da informação

Considere que foi gerado código executável para o programa abaixo.

```

1  .section .data
2  A: .quad 0
3  .section .text
4  .globl _start
5  _start:
6      movq $0x123456789abcdef,%rcx # instrucao 1
7      movq %rcx, A                 # instrucao 2
8      movq $60, %rax
9      movq $13, %rdi
10     syscall

```

A parte que iremos explorar são as duas instruções comentadas.

A instrução da linha 6 copia uma constante (ou “valor imediato”) para o registrador `%rcx`. Durante a execução do programa usando o `gdb`, pode-se analisar o valor contido em `%rcx` após a execução desta instrução:

```

...
rcx      0x123456789abcdef      81985529216486895
...

```

A primeira coluna mostra o registrador. A segunda indica seu valor hexadecimal e a última o seu valor em decimal que é obtido pela fórmula abaixo.

$$(123456789ABCDEF)_{16} \quad (C.1)$$

$$(F)_{16} \times 16^0 + (E)_{16} \times 16^1 + (D)_{16} \times 16^2 + (C)_{16} \times 16^3 + \dots + (2)_{16} \times 16^{13} + (1)_{16} \times 16^{14} \quad (C.2)$$

$$= (15)_{10} \times 16^0 + (14)_{10} \times 16^1 + (13)_{10} \times 16^2 + (12)_{10} \times 16^3 + \dots + (2)_{10} \times 16^{13} + (1)_{10} \times 16^{14} \quad (C.3)$$

$$= (81985529216486895)_{10} \quad (C.4)$$

Nesta fórmula, a Equação C.2 indica que cada “dígito” hexadecimal é multiplicado por um valor na base 16. A Equação C.3 indica a mesma multiplicação, porém com os valores hexadecimais substituídos por seus decimais correspondentes. Por exemplo, como o dígito hexadecimal F corresponde ao decimal 15, temos:

$$(F)_{16} = (15)_{10}$$

A Equação C.4 é o valor decimal correspondente, ou seja:

$$(123456789ABCDEF)_{16} = (81985529216486895)_{10}$$

Quanto mais à esquerda estiver o dígito de um número, mais “significativo” ele é. Observe que o dígito F da representação hexadecimal é aquele que é multiplicado por $16^0 = (1)_{10}$ enquanto que o dígito 1 é multiplicado por $16^{14} = (72057594037927936)_{10}$. Isto implica dizer que, neste número, o dígito 1 é mais significativo que o dígito F em função da posição que ele ocupa no número hexadecimal (apesar de que F é maior que 1).

Existem duas formas de armazenar números hexadecimais na memória, chamados “big-endian” e “little-endian”.

little-endian Os dígitos mais significativos são armazenados nos endereços menores de memória;

big-endian Os dígitos mais significativos são armazenados nos endereços maiores de memória.

Esta taxonomia curiosa tem origem no livro “As Viagens de Gulliver”, onde havia uma guerra entre Lilliput e Blefuscu. O motivo da guerra era que os moradores de Lilliput abriam ovos cozidos no lado “menor” dos ovos (little-endian) enquanto os moradores de Blefuscu os abriam no lado maior (big-endian)¹.

Como exemplo, considere o número hexadecimal $(0A0B0C0D)_{16}$ ou $0x0A0B0C0D$.

Uma forma de visualizar as duas formas de representação é através da tabela abaixo:

Memória	little-endian	big-endian
M[A]	0x0A	0x0D
M[A+1]	0x0B	0x0C
M[A+2]	0x0C	0x0B
M[A+3]	0x0D	0x0A

A arquitetura AMD64 usa a representação *little-endian*. Por esta razão, a visualização de números no depurador poderia ficar invertida.

Considerando que a memória “cresce” para a direita, estes números seriam vistos da seguinte forma:

little-endian $0x0D0C0B0A$

big-endian $0x0A0B0C0D$

Alguns computadores representam números hexadecimais usando o método little-endian enquanto que outros usam o big-endian.

Como o AMD64 representa little-endian, a representação de números na memória segue o padrão normal. Porém algumas ferramentas imprimem o conteúdo da memória do maior para o menor, confundindo um pouco.

Como exemplo, considere a simulação abaixo:

```
> gdb littleBigEndian
...
(gdb) set disassemble-next-line on
(gdb) break _start
(gdb) run
Breakpoint 1, _start () at littleBigEndian.s:7
7      movq $0x0123456789ABCDEF, %rax
=> 0x0000000004000b0 <_start+0>:      48 b8 ef cd ab 89 67 45 23 01      movabs $0x123456789abcdef,%rax
8      movq $1, B
=> 0x0000000004000ba <_start+10>:      48 c7 04 25 e9 00 60 00 01 00 00 00      movq $0x1,0x6000e9
9      movq %rax, A
=> 0x0000000004000c6 <_start+22>:      48 89 04 25 e5 00 60 00      mov %rax,0x6000e5
```

¹ Cultura inútil também é cultura.

A representação desta instrução na memória ocorre como indicado abaixo.

Memória	little-endian
M[0x4000b0]	0x48b8
M[0x4000b2]	0xefcd
M[0x4000b4]	0xab89
M[0x4000b6]	0x6745
M[0x4000b8]	0x2301
M[0x4000bA]	0x48c7
M[0x4000bC]	0x0425
M[0x4000bF]	0xe900
...	

Observe onde e como foi representada a constante 0x0123456789ABCDEF entre os endereços 0x4000b2 e 0x4000b9.

Agora vamos passar à instrução 9 (`movq %rax, A`). O simulador mostra (à direita) que o endereço efetivo de A começa em 0x6000e5 e ocupa oito bytes.

Usando o simulador, é possível ver como é a representação na memória. Neste caso, basta imprimir oito bytes a partir do endereço (x/8b 0x6000e5). Porém, vamos imprimir 16 bytes só para mostrar que a memória cresce para baixo.

```
(gdb) s
9      movq %rax, A
=> 0x0000000004000c6 <_start+22>:      48 89 04 25 e5 00 60 00      mov    %rax,0x6000e5
10     movq B, %rbx
=> 0x0000000004000ce <_start+30>:      48 8b 1c 25 e9 00 60 00      mov    0x6000e9,%rbx
(gdb) x/16b 0x6000e5
0x6000e5:      0xef 0xcd 0xab 0x89 0x67 0x45 0x23 0x01
0x6000ed:      0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02
(gdb)
```

Como a representação do AMD64 adota o modelo little-endian, não é surpresa que os bytes sejam apresentados “invertidos” (0xef no menor endereço e 0x01 no maior).

Esperamos minimizar as confusões que irão certamente ocorrer quando houverem visualizações de conteúdos na memória.

E para evitar mais confusões, não vamos explicar o que ocorreria se a representação de dados na memória fosse apresentada do maior endereço (em cima) para o menor endereço (em baixo). Afinal, a negação de uma negação é uma afirmação.

Se o objetivo é analisar somente o conteúdo da seção `.data`, o gdb disponibiliza o comando `print` que permite duas formas principais:

print <rótulo> imprime o conteúdo do rótulo indicado.

print &<rótulo> imprime o endereço do rótulo indicado.

C.3 Outros comandos de interesse

O GDB disponibiliza vários outros comandos, sendo que dois se destacam para este livro:

set environment O GDB não executa o programa, mas sim o emula. Por esta razão, ele não herda as variáveis de ambiente, como HOME, PWD, LD_LIBRARY_PATH, entre outros. Para atribuir valores para estas variáveis de ambiente, utiliza-se este comando. Por exemplo:

```
(gdb) set environment LD_LIBRARY_PATH=.
```

disassembly <função> este comando imprime o conteúdo da função indicada em assembly. Por exemplo:

```
(gdb) disas printf
Dump of assembler code for function printf:
0x00007ffff7867340 <+0>:      sub    $0xd8,%rsp
0x00007ffff7867347 <+7>:      test   %al,%al
```

```
0x00007fff7867349 <+9>:      mov    %rsi,0x28(%rsp)
0x00007fff786734e <+14>:     mov    %rdx,0x30(%rsp)
...
```

D. Memória Virtual

É possível que programas sejam maiores do que a memória física. Por exemplo, um programa que precise de 2Gbytes em um computador que só tem 1Gbytes de memória física. Este problema não é novo, desde 1956[39] várias soluções foram tentadas.

Apesar de existirem várias soluções para este problema, este capítulo descreve resumidamente duas soluções que, de alguma forma, foram utilizadas em outras partes do texto.

A primeira solução, apresentada na Seção D.1 é chamada *overlay*. A segunda solução é apresentada na Seção D.2 é chamada paginação.

D.1 *Overlay*

Aqui, a ideia é dividir o programa em pedaços, chamados *overlays*, e carregar cada um deles na memória individualmente, removendo outros se for o caso.

Para dar uma ideia melhor de como funciona esta técnica, descrevo uma situação que era bastante comum na década de 80. Eu era programador de uma empresa, e os computadores da época tinham muita pouca memória (quando comparado com os atuais), que variavam entre 128Kbytes e 512Kbytes. As aplicações mais comuns eram aquelas que disponibilizavam três opções para o usuário: inclusão, alteração e exclusão.

O usuário que usava este tipo de programa sempre estava em uma das seguintes situações:

1. na tela que pergunta qual das três operações o usuário quer executar,
2. nas telas referentes à inclusão,
3. nas telas referentes à alteração,
4. nas telas referentes à exclusão.

O importante a ser observado é que cada situação exclui as demais, ou seja, quando o usuário estiver na parte de inclusão, ele não usará o código referente à alteração e nem à exclusão. O mesmo ocorre quando estiver em alteração (que não precisa de inclusão e nem de exclusão) ou exclusão (que não precisa de inclusão e nem de alteração).

Para dar um toque mais “palpável” a este cenário, considere a Figura 38. Do lado esquerdo da figura, apresentamos o programa indicado. O programa é composto pelos módulos de Inclusão, Alteração, Exclusão e a tela que pede a opção do usuário.

A título de exemplo, considere que o módulo de inclusão ocupa 100 bytes, o módulo de alteração ocupa 110 bytes, o módulo de exclusão ocupa 60 bytes e as opções ocupam 90 bytes.

Do lado direito da figura, vemos que a memória física tem 150 bytes. Porém, o programa todo precisaria de $100 + 110 + 60 + 90 = 360$ bytes. Em outras palavras, este programa não “cabe” na memória.

Porém uma análise mais cuidadosa mostra que a inclusão, que precisa de 110 bytes, cabe sozinha na memória. O mesmo ocorre com a alteração, com a exclusão e com as opções. Aliás, os módulos de exclusão e de opções cabem juntos.

Como era comum este tipo de situação (programas maiores do que a memória), as linguagens de programação muitas vezes tinham diretivas adicionais para que fossem definidos os blocos que poderiam ser carregados na memória simultaneamente, os *overlays*.

A ideia era escrever, em alguma parte do código, algo como `BEGIN_OVERLAY` e `END_OVERLAY` para dizer ao compilador quais trechos de código correspondiam a cada *overlay*.

Desta forma era o programador quem especificava cada *overlay*. Não era muito divertido quando um *overlay* não cabia na memória. Era necessário “compactar” o código de alguma forma, como por exemplo, trocando sequências de código longas, porém claras, por sequências equivalentes pequenas, porém nada claras.

A ideia desta solução é basicamente empurrar o problema para o programador. Ele que deveria separar o programa em *overlays*. Porém, foi uma solução muito utilizada (em especial por este que escreve o texto).

Apesar de parecer arcaica e sem propósito nos dias atuais, esta solução é muito parecida com a forma de se gerenciar bibliotecas dinâmicas, onde o programador é quem diz quais são as funções que devem ser mapeadas na memória a cada passo. Em vários casos, o programador é capaz, inclusive de saber qual o uso de memória no pior caso (ou seja, quando os objetos dinâmicos que estão presentes na memória física simultaneamente ocupam a maior quantidade de espaço).

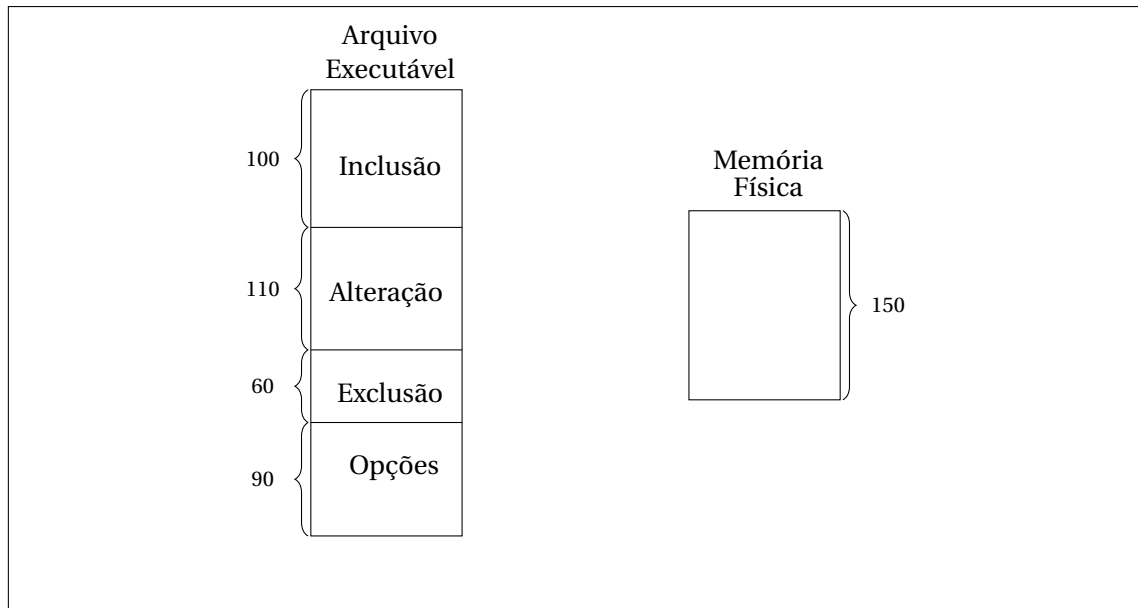


Figura 38 – Todos os segmentos não cabem na memória física.

D.2 Paginação

A paginação é a forma que grande parte dos sistemas operacionais modernos usam para fazer com que programas maiores do que o total de memória física possam ser executados.

Um dos pilares do seu sucesso é que nem todo programa precisa estar na memória para ele executar, pois:

1. é argumentado que a maior parte do tempo de execução de um programa é gasto em pequena parte do código.
2. o princípio da localidade [33] diz que os programas tendem a reutilizar dados e instruções usados recentemente.

Um dos maiores atrativos para o uso deste mecanismo é que ele não precisa ser gerenciado pelo programador. Quem gerencia o processo é o Sistema Operacional, ajudado por uma ferramenta de *hardware* chamada MMU (Memory Management Unit).

Considere a Figura 39, baseada em [43]. Esta figura descreve um sistema onde a memória virtual permite um espaço de endereçamento de 64K bytes, enquanto que a memória física tem somente 32K bytes. O espaço de endereçamento virtual pode conter qualquer coisa, e no contexto deste texto, contém um processo (a caixa de areia na qual um processo pode “brincar”). Um processo não pode ser executado a partir da memória virtual, então, é necessário copiá-lo primeiro para a memória física para somente então poder executá-lo.

O lado esquerdo da Figura 39 apresenta o espaço de endereçamento virtual, enquanto que o lado direito apresenta o espaço de endereçamento real. Tanto um quanto outro está dividido em blocos de 4Kbytes, chamados páginas. São 16 páginas para endereços virtuais (páginas virtuais 0 até 15) e 8 páginas de endereços físicos (páginas reais 0 até 7).

Agora, vamos analisar o que ocorre ao acessar endereços virtuais. Suponha que o espaço de endereçamento virtual contém um programa em execução (um processo). Como já é conhecido, quando um programa é executado, ele acessa instruções que estão na memória física. Como cada página virtual é composta por 4Kbytes, ela é capaz de conter várias instruções. A título de exemplo, considere que a execução do processo exige a leitura de instruções que estão na página virtual 01. O processo segue o fluxo das instruções e executa várias instruções nesta página. Em algum momento, ele acessa a página seguinte, a página virtual 02. Esta página contém instruções que acessam variáveis globais (que estão na página virtual 04). Em seguida, faz uma chamada de procedimento. Este procedimento está na página virtual 09 e os seus parâmetros estão na página virtual 12. Ao terminar o procedimento, o processo volta para a página virtual 02 e finaliza a sua execução.

Ao longo de toda a vida do processo, ele acessou as seguintes páginas virtuais: 01, 02, 04, 09, 12 e 02.

Sabendo agora quais páginas virtuais foram acessadas, vamos abordar como é feito o mapeamento entre páginas virtuais e físicas.

Quando um processo inicia a execução, é iniciada uma tabela de mapeamento de páginas virtuais para páginas físicas. Inicialmente, esta tabela é vazia, como mostrado na Tabela 23.

Espaço de Endereçamento Virtual			Espaço de Endereçamento Real		
Endereços Virtuais	Página Virtual	Número Página Virtual	Número Página Real	Página Real	Endereços Reais
00-04K		00	00		00-04K
04-08K		01	01		04-08K
08-12K		02	02		08-12K
12-16K		03	03		12-16K
16-20K		04	04		16-20K
20-24K		05	05		20-24K
24-28K		06	06		24-28K
28-32K		07	07		28-32K
32-36K		08			
36-40K		09			
40-44K		10			
44-48K		11			
48-52K		12			
52-56K		13			
56-60K		14			
60-64K		15			

Figura 39 – Modelo de Memória Virtual

A tabela é composta por 16 linhas (uma para cada página virtual). Todas as páginas inicialmente estão marcadas com “X”, pois não estão mapeadas em nenhuma página física.

A primeira página virtual acessada é a página 01. Como esta página não está mapeada em memória física, ocorre uma armadilha (trap) chamada “page fault”. O nome armadilha é apropriado. A ideia é que se alguém tentar acessar algo que não pode, a armadilha chama o sistema operacional para gerenciar o acesso. Neste momento, o sistema operacional pode verificar se o usuário tem permissão para acessar aquela página ou não. Quando não tem, normalmente o programa é abortado com a simpática mensagem “segmentation fault”. Quando tem permissão, o sistema operacional tem mais trabalho.

Primeiro ele verifica se aquela página corresponde alguma página já mapeada por outro processo, como por exemplo, a página de um objeto compartilhado ou dinâmico (Figura 27) ou do sistema operacional (Figura 14). Em caso positivo, usa aquela mesma página física. Em caso negativo, escolhe uma página física para onde copiar a página virtual de acordo com algum critério. Vários critérios de escolha podem ser usados, como por exemplo, escolher a página física que está há mais tempo sem acessos.

Suponha o segundo caso, e que a página física que o sistema operacional escolheu foi a página física 02 para conter a página virtual 01. Com isso, nova tabela de mapeamento é alterada, como indicado na Tabela 24. Nesta nova tabela, somente a entrada 1 foi alterada.

Isto implica dizer que o processo acredita que está usando a página virtual 01, mas na realidade esta página está mapeada na página real 02, ou seja, ele está usando a página real 02. É importante observar que se este processo for colocado em execução novamente, a página virtual 01 pode ser mapeada em outra página física.

Neste ponto, acredito que os leitores devem estar se perguntando como é que um endereço virtual pode ser mapeado em endereço físico. A explicação mais detalhada será apresentada na Seção D.2.1. Por enquanto basta acreditar que funciona.

Página Virtual	Página Física
1	X
2	X
3	X
4	X
5	X
6	X
7	X
8	X
9	X
10	X
11	X
12	X
13	X
14	X
15	X

Tabela 23 – Tabela de mapeamento de endereços virtuais para endereços físicos: situação inicial

Página Virtual	Página Física
1	2
2	X
3	X
4	X
5	X
6	X
7	X
8	X
9	X
10	X
11	X
12	X
13	X
14	X
15	X

Tabela 24 – Tabela de mapeamento de endereços virtuais para endereços físicos: Após acesso à página virtual 7

Conforme visto antes, o processo acessa as páginas na sequência. Assim, após trabalhar na página virtual 01, a próxima página virtual a ser acessada é a página virtual 02.

Quando o processo tentar acessar a página virtual 02, ocorrerá outra armadilha, que fará com que o sistema operacional seja chamado. Digamos que desta vez, o SO determina que a página real 05 seja usada para conter a página virtual 02. O mesmo ocorre com as outras páginas virtuais à medida que vão sendo utilizadas. Ao final, a tabela de mapeamentos fica como indicado na Tabela 25.

É importante observar que o processo todo é controlado pelo Sistema Operacional sem necessidade de intervenção do usuário.

Uma pergunta natural aqui é como que este mecanismo se comporta quando existem vários processos em execução simultânea.

Para explicar este cenário, considere que dois processos, processo “A” e processo “B” estão em execução simultânea. Cada processo tem a sua tabela de mapeamento de endereços virtuais para endereços físicos.

Página Virtual	Página Física
01	02
02	05
03	X
04	07
05	X
06	X
07	X
08	X
09	01
10	X
11	X
12	04
13	X
14	X
15	X

Tabela 25 – Tabela de mapeamento de endereços virtuais para endereços físicos: Após acesso às páginas virtuais 01, 02, 04, 09, e 12

Considere agora que as páginas virtuais do processo “A” estão mapeadas nas páginas físicas 00, 01, 02, 03 e 04. O processo “B” tem páginas virtuais mapeadas nas páginas físicas 05, 06, e 07. Observe que todas as páginas físicas estão ocupadas.

O processo “B” está em execução e acessa uma página virtual que não foi mapeada em memória física. Como já foi explicado, a tentativa de acessar a página virtual não mapeada em página física gera uma armadilha. Esta armadilha desvia o fluxo para o sistema operacional, que é incumbido da tarefa de mapear aquela página virtual em alguma página física.

Como todas as páginas físicas estão ocupadas, o sistema operacional terá de liberar alguma página física para conter a página virtual. O sistema operacional observa que a página física que está sem uso há mais tempo é a página 03. Sendo assim, ele:

- copia o conteúdo da página física 03 em disco. Digamos, para o arquivo “/swap/xxx”;
- vai na tabela de mapeamento do processo “A” e indica que aquela página virtual (digamos página virtual 12) não está mais mapeada na página física 03, mas sim copiada em disco.
- copia a página virtual de “B” para a página física 03.
- passa o controle para o processo “B”.

Após tudo isto, a execução volta ao processo “B”.

Em algum momento, o sistema operacional irá retirar o processo “B” da execução, e eventualmente, o processo “A” ganhará controle da CPU. Quando o fizer, é possível que tente acessar a página virtual 12, que estava mapeada na página física 03. Porém, ao tentar acessar aquela página, ocasionará um page fault pois aquela página física não está associada ao processo “A”.

A armadilha chamará o sistema operacional, que por sua vez terá de encontrar uma página física para abrigar a página virtual 12 de “A”. Ao encontrar uma página candidata, digamos página real 07, o sistema operacional irá copiar o conteúdo de “/swap/xxx” para a página física 07.

A página física 07 é exatamente igual ao que era a página física 3 antes de ser salva em disco, e o processo “A” nem sequer percebe a diferença.

O termo “área de swap” é usado para designar o local do disco onde as páginas removidas da memória física são armazenadas. No linux, esta área é criada na instalação do sistema em uma partição específica chamada *SWAP Partition*.

D.2.1 MMU

Em um sistema operacional com paginação, um processo comum jamais sabe quais os endereços da memória física está utilizando. O único que tem este privilégio é o sistema operacional. Um processo comum tem somente acesso a endereços da memória virtual. A tradução de endereços de memória virtual para memória física é feita pela MMU. Este

dispositivo está embutido na CPU, e a função dele é receber um conjunto de bits contendo um endereço virtual e fazer conversão para o endereço físico. A maneira com que a MMU faz isso é o objeto desta seção.

Considere que o rótulo `_start` está no endereço $(7120)_{16}$. Este endereço está na página virtual 7. todos os endereços entre $(7000)_{16}$ e $(7999)_{16}$ estão na página virtual 07.

Quando o programa é iniciado, o fluxo de execução desvia para o rótulo `_start`. Neste momento, o registrador program counter (PC) conterà o valor $(7120)_{16}$. Suponha que esta página virtual foi mapeada para a página física 02. A MMU irá converter o endereço virtual $(7120)_{16}$ em endereço físico $(2120)_{16}$ somente quando o endereço for colocado no barramento que conecta a CPU e a memória.

A Figura 40 exemplifica esta conversão. Como a figura mostra, o endereço de entrada é $(7120)_{16}$. Observe que no exemplo, o endereço de entrada contém desesseis bits, enquanto que o endereço de saída contém 15 bits.

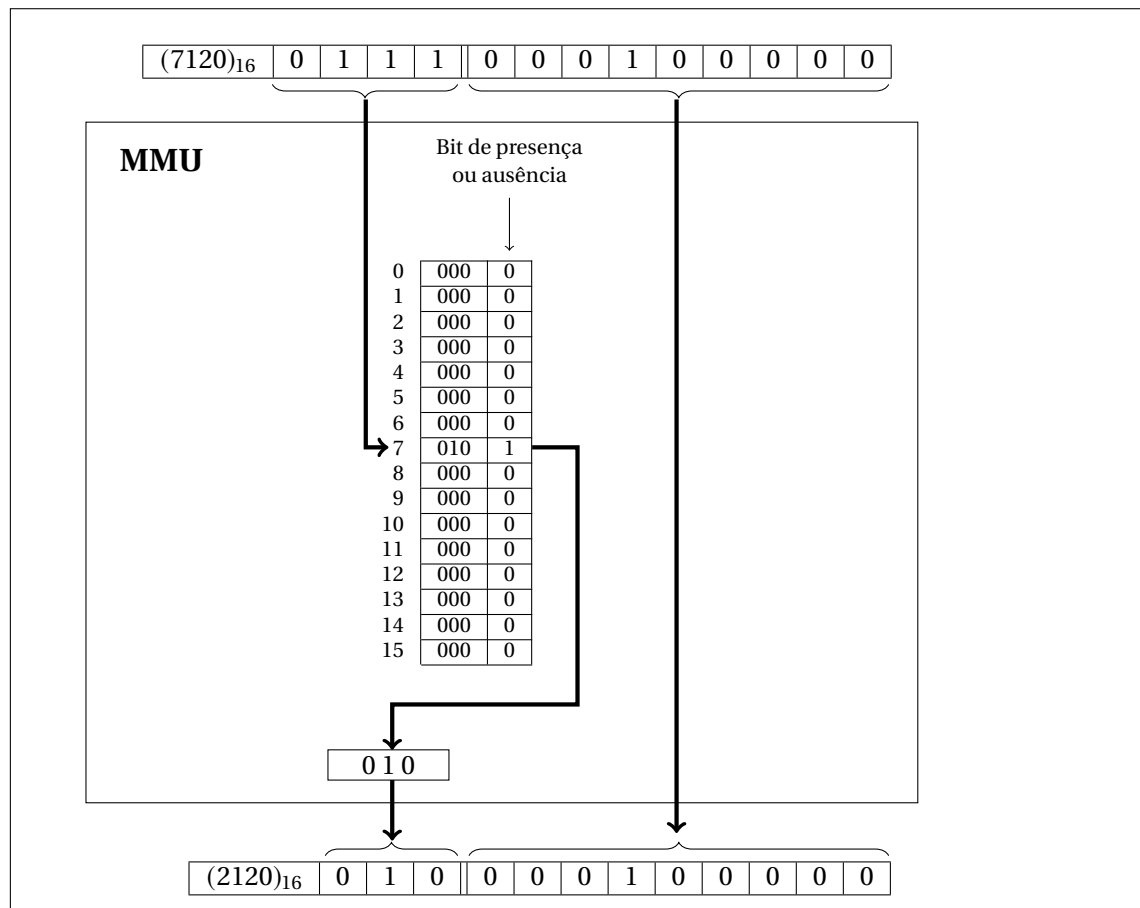


Figura 40 – MMU convertendo endereço virtual $(7120)_{16}$ para endereço físico $(2120)_{16}$.

O que a MMU faz é basicamente:

1. converter os quatro primeiros bits (número da página virtual) para três bits (número da página física);
2. copiar os 12 bits restantes, sem alterá-los.

Para fazer a primeira parte, a MMU utiliza uma tabela, chamada tabela de páginas. Esta tabela de páginas é a implementação da tabela mapeamento apresentada nas Tabelas 23, 24 e 25. Para comprovar, compare a Figura 40 e a Tabela 23.

Vamos agora analisar um pouco melhor alguns aspectos do funcionamento da tabela de páginas apresentada na Figura 40.

O modelo de paginação leva em consideração que os endereços virtuais de 16 bits foram divididos em duas partes: uma parte indica o número da página, e a outra indica o deslocamento dentro da página. No exemplo, o número da página ocupa 4 bits, para um total de $2^4 = 16$ páginas virtuais e o deslocamento ocupa 12 bits para um total $2^{12} = 4096 = 4\text{Kbytes}$ endereços dentro de cada página.

Vamos explorar um pouco mais este aspecto. A Tabela 26 corresponde à enumeração de todos os endereços virtuais possíveis usando 16 bits.

Endereço	Número da Página Virtual	Deslocamento
$(0000000000000000)_2$	0	0
$(0000000000000001)_2$	0	1
$(0000000000000010)_2$	0	2
...		
$(0000111111111111)_2$	0	4095
$(0001000000000000)_2$	1	0
$(0001000000000001)_2$	1	1
$(0001000000000010)_2$	1	2
...		
$(1111111111111111)_2$	15	4095

Tabela 26 – Relação entre endereços e as páginas

Os endereços estão em sequência. Assim, a primeira linha pode ser lida usando-se o número binário ou o par [número da página virtual, deslocamento]. Como exemplo, veja que o endereço virtual $(0001000000000000)_2$ (Figura 40) indica a página 1, deslocamento 0.

Considere agora uma sequência de endereços em uma mesma página virtual. Digamos $(7120)_{16}$, $(722F)_{16}$, $(7450)_{16}$. Todos estes endereços estão na página virtual 07.

O nosso exemplo diz que esta página virtual foi mapeada para a página física 02, ou seja, há uma cópia do conteúdo da página virtual 07 na página real 02. Pela MMU, sabemos que os endereços virtuais serão mapeados para os seguintes endereços: $(2120)_{16}$, $(222F)_{16}$, $(2450)_{16}$.

Lembrando que o primeiro byte da página virtual 7 está copiado no primeiro byte da página virtual 2, que o segundo byte da página virtual está copiado no segundo byte da página virtual, podemos ver que o conteúdo do endereço $(120)_{16}$ da página virtual 7 está copiado no endereço $(120)_{16}$ da página real 2.

Assim, o que a MMU faz é basicamente dividir o endereço em página + deslocamento, e converter o número da página virtual em número de página real.

Como a MMU converte todos os endereços que estão na CPU (endereços virtuais) para endereços físicos na memória (e muito rápido, como pode ser observado), temos uma situação onde ao olhar o valor dos registradores da CPU vemos endereços virtuais, e um pouco antes de estes valores serem colocados no barramento da memória, eles são convertidos para endereços físicos como mostrado na Figura 41.

D.2.1.1 Mecanismos de Proteção

A Figura 40 apresenta um bit de presença ou ausência que indica se aquela página virtual está ou não mapeada na memória real.

Este bit contém informação gerencial, e não é difícil imaginar a inclusão de outros bits armazenando outras informações gerenciais sobre cada página.

O projeto de muitas MMUs inclui bits para proteção. Por exemplo, a MMU do Alpha ALX - TLN 21064 inclui os seguintes bits [16]:

- Habilitado para leitura do usuário;
- Habilitado para escrita do usuário;
- Habilitado para leitura do kernel;
- Habilitado para escrita do kernel;

Uma combinação entre estes bits pode habilitar acessos diferentes ao mesmo espaço de endereçamento. Por exemplo, não é difícil ver como fazer para que as páginas contendo o sistema operacional só possam ser acessadas quando o processo estiver em modo supervisor (numa chamada ao sistema).

Como o foco é mostrar como funcionam os mecanismos de segurança, simplificamos um pouco as coisas por questões didáticas. Nesta simplificação, indicamos somente três bits de proteção. Estes bits indicam se os processos têm permissão de leitura, escrita e execução em cada uma das páginas.

Usaremos a sigla *rwX* para referenciar estes bits². Assim, as proteções de *rwX*(*PV0_i*) = 100 (permite leitura, não permite escrita ou execução).

A última coluna de cada tabela mostra em qual página física aquela página virtual foi mapeada. Assim, a *PV1_i* está mapeada na página física 1 (*PF1*), e o processo pode ler e escrever naquela página, mas não pode executar código. Esta página poderia corresponder à região da *stack*, *heap* ou *data*, mas não à região *text*.

Considere que o processo *P_i* é um processo regular que é executado em modo usuário e que o processo *P_j* é executado em modo supervisor. Estes dois processos compartilham a página física 6 (*PF6*), porém com permissões diferentes. Enquanto o processo *P_j* pode ler, escrever e executar *PF6*, o processo *P_i* pode somente ler.

Como curiosidade, vamos observar o ponto de vista do processo *P_i*. Suponha que ele está em execução e que ele examina uma variável mapeada em *PV0_i*. Ele é retirado de execução e o processo *P_j* continua. Suponha que ele escreve um valor no endereço virtual *PV2_j*, que mapeia na mesma página física de *PV0_i*. Esta escrita altera o valor da variável citada no processo *P_i*. Quando este processo continua a execução, subitamente percebe que a variável foi modificada. Do ponto de vista de *P_i* é como se um coelho se materializasse do nada.

Um exemplo mais completo apresentaria dois conjuntos *rxw*: um usado quando o processo estiver em modo usuário (*rxw_u*) e outro em modo supervisor (*rxw_s*). Suponha que o processo *P_i* esteja executando em modo usuário. Neste caso, os bits usados são *rxw_u*. Seja *PVX_i* uma página virtual que corresponde à área do sistema operacional. Então *rxw_u*(*PVX_i*) = 000, ou seja, não pode acessar a página. Porém, quando ocorre uma chamada ao sistema, o processo muda para o modo supervisor, e os bits usados poderiam passar a ser *rxw_s* = 111. Ou seja, o mesmo espaço virtual de endereçamento pode ser usado para acessar o sistema.

Exercícios

D.1 Como explicado na página 199, quando uma página real é associada a uma página virtual é possível usar uma página real já utilizada por outro processo ou substituir uma já existente. Baseando-se na Figura 27,

- a) explique se é possível que a página real já esteja na memória caso:
 - i. a página virtual estiver na área de segmentos do processo.
 - ii. a página virtual estiver na área de segmentos dinâmicos.
 - iii. a página virtual estiver na área reservada ao sistema.
- b) explique se é possível que a página real não esteja na memória caso:
 - i. a página virtual estiver na área de segmentos do processo.
 - ii. a página virtual estiver na segmentos dinâmicos.
 - iii. a página virtual estiver na área reservada ao sistema.

D.2 É possível verificar as permissões das páginas virtuais de um processo no diretório */proc*. Para fazê-lo, é necessário primeiro descobrir o PID (Process ID) do programa:

- a) crie um programa em C contendo um ponto de parada, por exemplo, com o comando *scanf*;
- b) antes do *scanf*, inclua o seguinte: `printf ("PID=%d\n", getpid());`;
- c) compile o programa gerando o executável, digamos *main*;
- d) execute o programa. Após imprimir o número do processo, ele ficará aguardando a entrada de dados;
- e) Existe um diretório com este número no diretório */proc*. Supondo que o PID=16866, veja o diretório correspondente em */proc*:

```
> ls /proc/16866
attr          cpuset        latency        mountstats     personality    stat
autogroup      cwd           limits         net             projid_map     statm
auxv           environ       loginuid       ns              root           status
cgroup         exe           map_files      numa_maps      sched          syscall
```

² a sigla vem das iniciais de *read*, *write* e *execute*, usado em ambientes linux para indicar as permissões de acesso a arquivos

clear_refs	fd	maps	oom_adj	schedstat	task
cmdline	fdinfo	mem	oom_score	sessionid	timers
comm	gid_map	mountinfo	oom_score_adj	smaps	uid_map
coredump_filter	io	mounts	pagemap	stack	wchan

f) Dentro deste diretório, veja o arquivo maps:

```
> cat /proc/16866/maps
00400000-004ef000 r-xp 00000000 08:01 6422534 /bin/bash
006ef000-006f0000 r--p 000ef000 08:01 6422534 /bin/bash
006f0000-006f9000 rw-p 000f0000 08:01 6422534 /bin/bash
006f9000-006ff000 rw-p 00000000 00:00 0
.....
```

A primeira linha indica que o segmento de memória virtual entre 00400000-004ef000 é composto por páginas de leitura e execução (mas não escrita). Este segmento contém a seção `.text`, e as permissões estão apropriadas. Para mais informações, digite

```
> man 5 proc
```

D.3 Faça a mesma análise para cada segmento apresentado na lista.

D.4 Acrescente variáveis globais, bss, heap e na stack, fazendo o programa imprimir seus endereços. Confira sua localização no arquivo maps.

D.5 Confira também as páginas dos objetos compartilhados e do sistema operacional.

Referências

- [1] Andrew Tannenbaum and Albert Woodhull. *Computer Networks*. Prentice Hall, 1981.
- [2] Andrew Tannenbaum and Albert Woodhull. *Operating Systems Design and Implementation*. Prentice Hall, 1987.
- [3] Jonathan Bartlett. *Programming From The Ground Up*. Bartlett Publishing, 2004.
- [4] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. https://course.ccs.neu.edu/csu4410/spim_documentation.pdf.
- [5] Benjamin Chelf. Building and using shared libraries.
- [6] Chongkyung Kil, Jinsuk Jun, and Christopher Bookholt. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. *Computer Security Applications Conference*, 2006.
- [7] Portable Applications Standards Committee. *Standard for Information Technology — Portable Operating System Interface: Rationale (Informative)*. IEEE Computer Society, 2001.
- [8] Tool Interface Standards Committee. *Executable and Linking Format (ELF)*. Linux Foundation, Maio 1995. Pode ser encontrado a partir de <http://refspecs.freestandards.org/>.
- [9] Jonathan Corbet. On vsyscalls and the vdso. *LWN.net*, 2011. <https://lwn.net/Articles/446528/>.
- [10] Microsoft Corporation. *Microsoft Portable Executable and Common Object File Format Specification*. O'Reilly Media, 2013. <https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547%28v=vs.85%29.aspx>.
- [11] Daniel Plerre Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2005.
- [12] Guilherme Herrmann Destefani. *Verificação Oportunista de Assinaturas Digitais para programas e bibliotecas em sistemas operacionais paginados*. 2005. Dissertação de Mestrado.
- [13] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Criel J.H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. Springer, 2 edition, 2012.
- [14] Gintaras R. Gircys. *Understanding and Using COFF*. O'Reilly Media, 1988.
- [15] Hector Marco-Gisber and Ismael Ripoll. On the effectiveness of nx, ssp, renewssp and aslr against stack buffer overflows. *13th International Symposium on Network Computing and Applications*, 2014.
- [16] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.
- [17] I.A.Dhotre and A.A.Puntambekar. *Systems Programming*. Technical Publications, 2008.
- [18] Intel, editor. *IA-32 Intel Architecture Software Developer's Manual Volume 3: General-Purpose and System Instructions*. Intel, 2019.
- [19] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, 2004.
- [20] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, 2005.
- [21] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1997.
- [22] Tomasz Kowaltowski. *Implementação de Linguagens de Programação*. Guanabara, 1983.
- [23] James R. Larus. *SPIM S20: A MIPS R2000 Simulator*. 1997. https://course.ccs.neu.edu/csu4410/spim_documentation.pdf.
- [24] John Launchbury. A natural semantics for lazy evaluation. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 07 2000.

- [25] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000.
- [26] M. V. Wilkes and W. Renwick. The edsac (electronic delay storage automatic calculator). *Mathematics of Computation*, 4, 1950. Artigos históricos do EDVAC disponíveis em <http://www.cl.cam.ac.uk/events/EDSAC99/reminiscences/>.
- [27] Ronald Mak. *Writing Compilers and Interpreters: A Modern Software Engineering Approach*. John Wiley & Sons, 3 edition, 2009.
- [28] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell, editors. *System V Application Binary Interface AMD64 Architecture Processor Supplement*. 2013. <http://x86-64.org/documentation/abi.pdf>.
- [29] Michaelis. *Michaelis Moderno Dicionário Da Língua Portuguesa*. Melhoramentos, 2004.
- [30] Oliver Mueller. Anatomy of a stack smashing attack and how gcc prevents it. *Dr. Dobbs's*, 2012.
- [31] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 2010.
- [32] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. *International Workshop on Memory Management*, 1995.
- [33] Peter J. Denning. The locality principle. *Communications of the ACM*, 48(7), 2005.
- [34] Peter Norton, Peter Aitken, and Richard Wilton. *The Peter Norton PC Programmer's Bible*. Microsoft Press, 1993.
- [35] P. J. Plauger. *The Standard C library*. Prentice Hall, 1992.
- [36] Ronald D. Reeves. *Windows 7 Device Driver*. Addison-Wesley, 2010.
- [37] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC Applied Algorithms and Data Structures series. Chapman and Hall/CRC, 1 edition, 2011. veja também <http://gchandbook.org/>.
- [38] Dennis M. Ritchie. The development of the c language. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 201–208, New York, NY, USA, 1993. ACM.
- [39] Laurie Robertson. Anecdotes. *IEEE Annals of the History of Computing*, 26(4), Oct.-Dec. 2004.
- [40] Robert C. Seacord. *Secure Coding in C and C++*. Addison-Wesley Professional, 2 edition, 2013.
- [41] Sirisara Chiamwongpaet and Krerk Piromsopa. The implementation of secure canary word for buffer-overflow protection. *IEEE International Conference on Electro/Information Technology*, 2009.
- [42] Andrew Tannenbaum. *Structured Computer Organization*. Prentice Hall, 1976.
- [43] Andrew Tannenbaum. *Modern Operating Systems*. Prentice Hall, 2001.
- [44] Ian Lance Taylor. A new elf linker. In *Proceedings of the GCC Developers' Summit*, 2008.
- [45] AMD64 Technology. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*. AMD, 2012.
- [46] AMD64 Technology. *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*. AMD, 2013.
- [47] Tim Lindholm and Alex Buckley Frank Yellin, Gilad Bracha. *The Java Virtual Machine Specification Java SE 8 Edition*. Addison-Wesley Professional, 2015.
- [48] Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4), 1986.
- [49] John von Neumann. *First Draft of a Report on the EDVAC*, volume 15. *IEEE Annals of the History of Computing*, October-December 1993.
- [50] David A. Wheeler. Program library howto. <http://tldp.org/HOWTO/Program-Library-HOWTO>.
- [51] David A. Wheeler. System v application binary interface. <http://www.x86-64.org/documentation/abi.pdf>.
- [52] J. Xu. Transparent runtime randomization for security. *22nd International Symposium on Reliable Distributed Systems*, 2003.