Tutorial Assembly Win32 x86

Introdução

O objetivo deste tutorial é ensinar Assembly para arquiteturas x86 (32 bits) sob o Sistema Operacional Windows. Embora este tutorial seja especificamente para Windows, eu gostaria de deixar claro que este tutorial é sobre Assembly para arquiteturas x86 de 32 bits (a nomenclatura para 64 bits é x86_64) e usa o Windows como "container" ou "host" por mera casualidade. Assembly não é o que posso chamar de flexível, pois temos que fazer muitas escolhas, isto é, há pouco espaço para generalidades. A linguagem Assembly pode ser diferente entre compiladores por causa das diferentes diretivas que cada um contém. Pode ser diferente entre processadores porque cada processador tem seu próprio conjunto de instruções. Pode ser diferente entre Sistemas Operacionais porque cada Sistema Operacional tem uma maneira diferente de controlar o acesso ao hardware. Esse texto mostra o Assembly dos processadores x86, utiliza o Windows e faz uso de um compilador conhecido como nasm (Netwide Assembler) com o auxílio de um software de ligação (linker) conhecido como alink. Você pode ler mais detalhes sobre o que realmente vamos estudar neste texto na seção "O que vamos estudar?".

A linguagem Assembly é muitas vezes reputada como difícil, complexa e extremamente poderosa. Eu não concordo com os dois primeiros adjetivos e tendo a discordar da expressão "extremamente poderosa", pois dentro do Windows todo mundo é igual, ou seja, os programas feitos diretamente em Assembly devem seguir as mesmas regras que um programa feito em Delphi. Nesse contexto o Assembly fica muito próximo de linguagens como Pascal ou C e perde muito de seu "poder". Para falar a verdade, é justamente o "poder" da linguagem Assembly que a torna praticamente inútil (peço aos que pularam da cadeira com a afirmação anterior que prestem mais atenção na palavra "praticamente" e menos atenção na palavra "inútil"). Cuidado! Eu vou repetir novamente: dentro do Windows. A linguagem Assembly fora de um Sistema Operacional terá tanto poder quanto o próprio processador, no sentido que de se for possível então será possível em Assembly. Por outro lado há linguagens que nem funcionam fora de um Sistema Operacional ou se funcionarem vai exigir praticamente a construção de um Sistema Operacional (construir em outra linguagem). Você pode encontrar maiores detalhes do que é Assembly na seção "O que é Assembly?".

Antes de começar a falar de Assembly eu gostaria de revisar três conceitos acho importante.

Dados e informação

Um computador é em essência uma máquina que ao entrarmos com um ou mais **dados** ela efetua algum **processamento** nesses dados e exibe uma ou mais **informações**. Os dados são informações em estado **bruto** dentro de um **determinado contexto**. As informações são os **resultados** de um processamento efetuado nos dados, isto é, são dados em um estado mais **polido** de acordo com um **determinado contexto**. Para ter uma ideia mais contextualizada do que é um dado, basta imaginar os dados fornecidos pelo IBGE (Instituto Brasileiro de Geografia e Estatística). Alguns dados do IBGE podem ser: número de pessoas, número de homens e de mulheres, taxa de natalidade ou taxa mortalidade, expectativa de vida ao nascer, etc.. Não

consigo imaginar algo muito específico agora, mas sei que vocês compreendem que estes dados podem ser úteis para vários setores da sociedade nas suas tomadas de decisões. Vejamos agora uma ideia mais contextualizada do que é uma informação. Podemos gerar uma informação (dado polidos) a partir do número de pessoas, número de mulheres e número de homens (dados brutos). Podemos determinar, por exemplo, que 80% das pessoas no Brasil são do sexo masculino, este dado é uma informação. No entanto, ao determinar a porcentagem de crianças do sexo masculino – 30% dos homens são crianças – a partir da porcentagem de pessoas do sexo masculino, observamos que o que antes era uma informação (80% das pessoas são do sexo masculino) agora passa a ser, neste contexto, um dado. Não sei se ofereci bons exemplos de dados que geram informações e de informações que se tornam dados gerando informações novas, mas acredito que você pegou a ideia e que você é capaz de pensar em exemplos melhores.

Entrada e saída

O computador precisa oferecer um meio para que os dados entrem. Em um computador pessoal de mesa (conhecido como desktop) é muito provável que exista um teclado onde podemos premer determinadas teclas e entrar com os dados de forma direta. Entretanto, é importante ter em mente que o computador não precisa necessariamente que os dados entrem de maneira tão explícita como o premir de uma tecla. Os computadores podem usar os mais diversos tipos de sensores para captar um determinado dado. Uma pesquisa pelo termo "sensor" na wikipedia vai mostrar uma grande variedade de sensores que podem ser usados para captura de dados. Seja qual for o dado de entrada ou o meio que ele "entra" no computador, este processo é facilitado por um **dispositivo de entrada.**

Até agora sabemos o que são dados e informações e que o computador oferece meios para que eles sejam inseridos através de algum dispositivo de entrada. No entanto, o computador também precisa mostrar o resultado do processamento efetuado nos dados que foram inseridos. Neste contexto é fácil entender que o monitor é um bom exemplo, pois através de um monitor conseguimos observar uma informação de maneira muito natural, isto é literalmente ver com os próprios olhos o resultado do que foi processado. Aqui podemos mais uma vez perceber a flexibilidade que um computador pode ter, pois a informação não precisa necessariamente ser mostrada por um dispositivo óptico. De maneira mais informal, não precisamos ver com os próprios olhos para ficarmos cientes do resultado de um processamento. Neste contexto qualquer informação que possa ser compreendia pelos órgãos do sentido é válida. Se você está em uma Central Nuclear e escuta uma sirene é muito provável que um computador obteve, através de um dispositivo de entrada, um dado de natureza radioativa, efetuou um processamento neste dado e emitiu a informação em um dispositivo de saída: a sirene. Neste caso a informação foi transmitida por um dispositivo sonoro e nem por isso é menos informativa. Apesar disto, alguns homens que trabalhavam na limpeza da Central Nuclear de Fukushima I (isso foi após o acidente trágico) não pensaram desta forma, pois ignoraram a informação que o computador transmitiu através de uma sirene e o resultado você já deve imaginar... Seja qual for a informação de saída ou o meio que ela "sai" do computador, este processo é facilitado por um **dispositivo de saída.**

Processamento

Sabemos que dados entram, informações saem, e vimos muitas vezes o verbo “processar”. O processamento de um dado tem o objetivo de gerar uma informação e é efetuado por um computador. No entanto, eu diria que é neste ponto em que as coisas começam a ficar “sérias” e onde os usuários se tornam usuários e os programadores se tornam programadores. Explico.

Do ponto de vista de um usuário os dados e as informações são o resultado de um processo efetuado pelo computador, como mencionei anteriormente. Esta percepção não esta errada, mas é deveras simplista. O programador, por outro lado, tem uma visão diferente e tende a separar o computador em partes cada vez menores de acordo com sua área de atuação. Este texto é para programadores que estão em um nível muito baixo no sentido de que a visão que ele tem é muito mais profunda que a visão de um usuário e um pouco mais profunda que a visão de um programador de nível mais alto. Colocando de maneira mais objetiva, o programador separa o computador em partes menores como dispositivos de armazenamento temporário ou permanente (memória principal, discos rígidos, etc.), dispositivos de entrada e saída de dados e informações (teclado, monitor, mouse, impressora, etc.) e dispositivos de processamento de dados (o próprio processador). Nós que estamos estudando uma linguagem de baixo-nível elevamos ainda mais o conceito de profundidade e dividimos os próprios processadores em componentes mais específicos e visualizamos os demais dispositivos também de maneira mais especializada. Além disso, adicionamos ao nosso conhecimento alguns componentes que são “obscuros” até mesmo para um programador de alto nível. Neste ponto nós começamos a dar mais importância aos componentes que transportam os dados ou informações de um lugar para outro dentro do computador. Estes dispositivos são conhecidos como barramentos. Essa “escadinha” no nível de profundidade não para por aqui (apesar de este texto parar). Um engenheiro pode ter uma visão bem mais profunda de um processador que um programador baixo-nível e consequentemente uma visão mais profunda que a abordada neste texto.

Eu quereria muito que você não produzisse um sentimento de superioridade por causa do que comentei acima. Um programador baixo-nível não é superior (superior aqui está num contexto ligado à complexidade de um conhecimento x comparado a um conhecimento y, não é no sentido moral, ou qualquer outro) a um programador de alto-nível só porque está estudando o computador de uma maneira mais profunda. Se você não concorda ou acredita que Assembly é mais difícil ou mais complexo que, por exemplo, Java, eu posso até entender. No entanto, vou fazer de tudo neste texto para mostrar a você que Assembly e todo o ambiente onde ele está imerso não merece a reputação que tem. De maneira mais direta, entenda que programação não se resume a memorizar um conjunto de regras de uma determinada linguagem que abstrai um determinado sistema. Programação é mais que isso, pois o fato de alguém saber ou estudar Assembly não faz dele um programador melhor. Eu, por exemplo, sei quase uma dezena de linguagens, mas este conhecimento quase não teve influência em minha habilidade na resolução de problemas computacionais. Você pode resolver um problema em um lenço de papel em uma lanchonete qualquer e transferir a solução para uma linguagem, seja ela Java ou Assembly. O importante aqui é lembrar que a linguagem foi apenas um meio intermediário para transformar a solução em um programa.

As figuras abaixo mostram como seria a visão que um usuário, um programador alto-nível e um programador baixo-nível, têm do computador. Entenda que isso não quer dizer que os usuários não podem conhecer maiores detalhes da máquina ou que os programadores “comuns” não podem se aprofundar mais nos detalhes do hardware. Em todo caso, isso implica que normalmente um programador baixo-nível precisa conhecer bem o hardware: a arquitetura do computador, o processador, o sistema operacional a memória, etc..

Figura 1 Possível visão de um usuário.

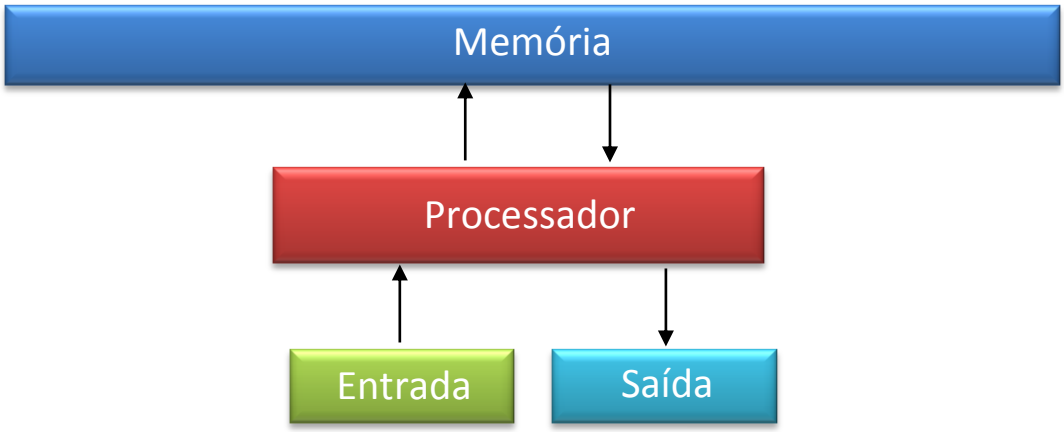

Figura 2 Possível visão de um programador alto-nível.

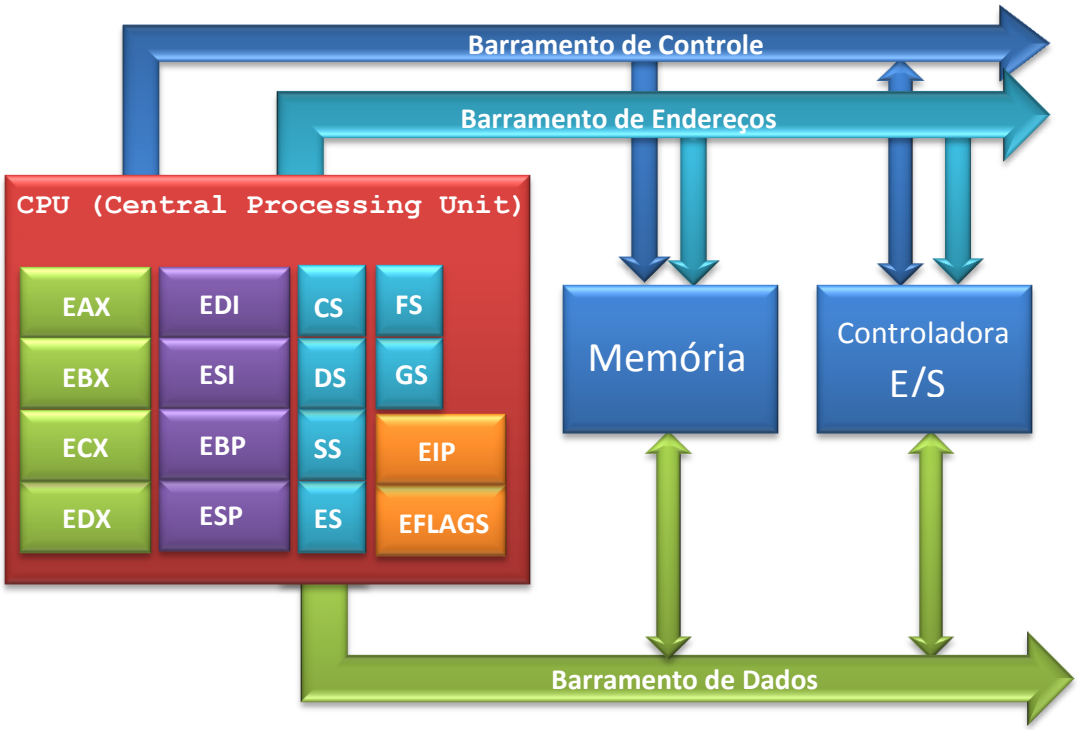

Figura 3 Possível visão de um programar baixo-nível.

O que vamos estudar?

Neste texto vamos deslocar nossa atenção principalmente para o processador. Obviamente não existe um único tipo de processador, então este texto vai comentar apenas os processadores conhecidos como x86. O termo x86 é um termo genérico que designa todos os processadores baseados em um processador antigo conhecido como 8086. Apesar do processador 8086 ser um produto da Intel e muitos dos x86 posteriores também serem, há outros fabricantes que também produzem processadores que recebem a designação x86. Um fabricante conhecido de processadores x86 é a AMD. Processadores x86 não é sinônimo de PC (PC é um termo que expressa o conceito de computador pessoal, mas na prática é aplicada a computadores baseados no IBM PC), pois não há nada que impeça um computador pessoal MacIntosh, da Apple, que geralmente não recebe a designação PC, usar um processador do tipo x86. Outro termo muito usado recentemente é o termo x86_64, que é nada mais que uma categoria de processadores x86 que utilizam registradores de 64 bits. Outro termo comum é o termo IA-32 que relaciona a arquitetura Intel para processadores de 32 bits e IA-64 que relaciona a arquitetura Intel para processadores de 64 bits. IA-32 é também um x86 e IA-64 é também um x86_64. Dito isto podemos definir melhor em qual ambiente vamos estudar.

Este texto se refere aos processadores x86 de 32 bits que estão espetados em um computador baseado em um IBM PC e que utiliza a Arquitetura de von Neumann. O termo ou expressão "Arquitetura de von Neumann" é até redundante neste contexto, mas se você não conhece esta expressão a sua pergunta deve ser: "O que é Arquitetura de von Neumann?", e não: "Eu tenho um processador x86 e um computador baseado em um IBM PC, será que ele também segue a Arquitetura de von Neumann?". A primeira pergunta será respondida neste texto e a segunda você vai perceber sozinho.

Nosso ambiente considera um processador x86, mas o foco deste texto não é o estudo específico do x86, mas sim o conjunto de instruções ou códigos operacionais ou ainda opcodes que esta família de processadores fornece para os programadores. Além do estudo dessas instruções e de seus mnemônicos ou apelidos, precisamos de um conhecimento razoável do ambiente onde o x86 está inserido. É por este motivo que foi preciso limitar o computador para apenas computadores PC. Vamos supor que um x86 esteja em um computador que não utilize teclado, monitor, mouse, etc.. Por mais que você saiba como ninguém o conjunto de instruções do x86 que é utilizado neste computador, é possível que você não saiba nem ligar este mesmo computador! Entendeu?

O que é Assembly?

Nosso ambiente já está definido. Já sabemos também que vamos estudar as instruções que os x86 disponibilizam para o programador. Programar utilizando as instruções do processador é programar em linguagem Assembly. O que difere é que a linguagem Assembly substitui os valores numéricos de cada instrução por uma sigla ou abreviação intuitiva que indica o que a instrução faz. Vamos supor que a nossa intenção seja colocar o número 10 na posição de memória cujo endereço é 100. Se a gente usar o código operacional (valor número da instrução) diretamente a gente teria um código como este:

1100 0110 0000 0110 0110 0100 0000 0000 0000 1010

Este valor está sendo representado com a base 2. O mesmo código na base 16 seria:

C6 0664 000A

E na base 10 seria:

850.510.741.514

O processador necessita exatamente deste valor para fazer o que a gente pretendia (colocar o número 10 na posição de memória cujo endereço é 100). No entanto, se a gente utilizar a linguagem Assembly esta mesma operação pode ser escrita da seguinte maneira:

MOV BYTE [100], 10

O código operacional da instrução que move um valor de um local para outro é representado na linguagem Assembly pelo mnemônico MOV. Na versão numérica da instrução os valores inseridos pelo programador (as constantes ou valores imediatos) acabam se misturando com os valores numéricos do código operacional causando uma confusão ainda maior. No entanto, até os menos experientes conseguem identificar, na versão em linguagem Assembly, os valores 100 e 10, e com o auxílio do mnemônico MOV podem até arriscar algum palpite do que está acontecendo. Os mnemônicos são muito intuitivos, basta espiar uma pequena lista deles para comprovar isto: MOV (mover), ADD (adicionar), SUB (subtrair), MUL (multiplicar), DIV (dividir), CALL (chamar), etc.. Não esqueça que eles são representados por palavras, ou abreviações, do idioma inglês.

Neste ponto é importante saber que algumas partes da instrução Assembly anterior não tem uma contraparte direta na instrução do processador. Na instrução anterior podemos destacar os colchetes (os colchetes indicam que o número 100 representa um endereço de uma célula de memória), a vírgula (a vírgula separa os operandos da instrução) e a palavra BYTE (BYTE indica que apenas a célula de memória de endereço 100 deve ser modificada, isto é, apenas um byte). Dependendo da quantidade desses caracteres auxiliares ou o uso de blocos de construção que substituem conjuntos inteiros de instruções pode tornar a linguagem Assembly uma linguagem de alto nível.

Seja como for, por mais que um código em linguagem Assembly seja bem fiel às instruções reais da máquina, este código pode ser executado pelo processador. A gente já viu que o processador exige a representação numérica da instrução no sistema numérico base 2 ou, como é mais conhecido, sistema binário. Desta forma faz-se necessário uma conversão do código em linguagem Assembly para um código em linguagem de máquina.

> ***Eu particularmente chamo este processo de "compilação" sem hesitar. O código contendo mnemônicos ou outros caracteres auxiliares eu chamo de "Linguagem Assembly" ou "Código Assembly", já o código que pode ser executado diretamente pelo processador eu não costumo ter uma preferência muito rígida por um ou outro termo, mas muitas vezes chamo de "Assembly". De qualquer forma, aparentemente não há um consenso neste sentido. Eu já li críticas sobre o uso do termo "compilar" relacionado com a "Linguagem Assembly" e indicando o termo "montar" e o termo "montador" para o programa que efetua este processo. Não sei de onde surgiu a palavra "montar", mas acredito***

que os termos “Assembly” e “Assembler” podem ser relacionados com reunião, reunir, compilação, compilar. Outros termos comuns incluem “Linguagem de Montagem”, “Linguagem de Máquina”, etc.. Em alguns chats é fácil receber críticas quando usamos os termos “Assembly” e “Assembler” de maneira intercambiável: “Asssembly é a linguagem e Assembler o programa que...”. O correto é sempre usar o idioma português, mas confesso que não é muito fácil manter-se no português, por vezes acabo usando o inglês Quem sabe uma hora eu revise este texto trocando, sempre que possível, os termos em inglês por um em português.

O programa responsável por pela conversão de um código em linguagem Assembly para um código em linguagem de máquina é conhecido como “Assembler”. A escolha do Assembler é um fator importante que afeta a maneira que vamos programar. Não há um padrão que define como a linguagem Assembly deve ser. Um código em linguagem Assembly destinado a ser compilado pelo GNU Assembler (GAS) é visivelmente diferente de um código destinado a ser compilado pelo Netwide Assembler (nasm). Curiosamente a diferença normalmente se restringe ao código Assembly, pois os resultados (o binário final ou o arquivo com as instruções Assembly convertidas em instruções numéricas) são muitas vezes idênticos. Este texto vai usar o Netwide Assembler (nasm).

Ainda no assunto “conversão”, lembre-se que até a representação em números binários é de certa uma abstração, pois no mundo real os valores são representados na memória por níveis diferentes de tensão assumindo sempre certa faixa de tensão para o valor 0 e outra faixa distinta de tensão para o valor 1 ou qualquer coisa do tipo. Em todo caso, eu não acho relevante conhecer os níveis exatos de tensão que representam o valor 0 ou o valor 1. Comentei sobre a representação física dos valores porque nem sempre eles são representados por níveis de tensão. Nos discos magnéticos (HD ou disco rígido ou até mesmo os lendários disquetes), por exemplo, o que vale é a direção dos polos positivos e negativos das moléculas da superfície magnética. Ainda não tive o prazer de me aprofundar em outras mídias como DVD-ROM, Blu-ray, HD-DVD ou CD-ROM, então deixo isto para o leitor mais curioso.

Você já viu o que é Assembly, mas talvez esteja pensando que é complicado ou que não está entendendo nada. Se for este o caso podes ficar tranquilo, pois eu estou apenas comentando alguns aspectos gerais e no decorrer do texto vamos retornar a todos os temas desta introdução com uma visão bem mais profunda, explicativa e prática.

Windows e Linguagem Assembly

Programar em Assembly é nada mais que combinar, de maneira engenhosa, um punhado de instruções. Uma linguagem de alto nível tem uma sintaxe bem mais rebuscada que a linguagem Assembly, pois em Assembly não há regras. Como assim não há regras? Já vimos que o processador tem uma coleção de instruções onde cada uma recebe um valor específico, um significado e um apelido. Porém não há regra muito rígida para usar estas instruções. Dada natureza da arquitetura ,o processador não tem muitos recursos para verificar se o seu programa está somando 2+2, como você imaginou, ou se em vez disso está formatando seu disco rígido. Você pode considerar que é pouco provável um programa que deveria somar 2+2 formatar o disco rígido. No entanto, imagine uma situação hipotética onde fosse possível colocar valores diretamente na memória principal (local onde o processador “pesca” suas

instruções – entenda “suas” como as instruções do processador, e não as SUAS instruções, pois curiosamente muitas vezes sentimos que são diferentes). Uma vez inserida as instruções, o processador vai executar as instruções como sempre faz. Ainda dentro da situação hipotética anterior, imagine que uma criança de 7 anos (aquelas que além de terem 7 anos também pintam o 7) coloque um punhado de valores aleatórios na memória principal e que, em uma coincidência cósmica, aqueles valores tome a forma de um programa capaz de formatar o disco rígido. Esta coincidência cósmica poderia impressionar e assustar você, mas a impassibilidade em que o processador fará o seu trabalho (pescar instruções da memória e executar, uma a uma) será ainda mais impressionante e ainda mais assustador! Em outras palavras, o processador não exige, por exemplo, que uma instrução JMP venha antes ou depois da instrução CMP. O processador também não se importa se o programa A está escrevendo na memória do programa B ou vice-versa.

Apesar da indiferença, o processador fornece instruções e mecanismos específicos que permitem que um programa A impeça o programa B, C, D, etc. de acessar tanto a memória uns dos outros quanto sua própria memória. Mais tarde veremos que um programa precisa necessariamente estar na memória principal para ser executado, então a partir desta informação, um programa que pode controlar o acesso indiscriminado da memória pode também controlar quem pode ou não executar e o que pode ou não ser executado. O programa A mencionado é geralmente conhecido como Sistema Operacional. Vamos conhecer um pouco mais sobre isso.

O sistema operacional escolhido neste texto é o Windows. O sistema operacional tem grande impacto sobre o nosso estudo porque uma vez que estamos “dentro” de um Sistema Operacional somos obrigados a respeitar suas regras. Não é interessante aprofundar muito neste assunto agora, então vou comentar apenas o necessário para entender as implicações da escolher um ou outro Sistema Operacional ou até mesmo a escolha de nenhum Sistema Operacional. Quando o computador é ligado o Sistema Operacional é iniciado (Windows no nosso caso) e com a ajuda direta do próprio processador ele restringe as opções de qualquer programa que ele julgar necessário. Esta restrição evita muitos inconvenientes envolvendo descuidos do programador ou programadores mal intencionados. O Windows não permite que um programa acesse a memória de outro programa ou que um programa formate o disco rígido e nem mesmo que um programa mostre uma reles mensagem do tipo “Alô mundo” na tela. No entanto, eu posso deduzir que você já viu um programa “Alô mundo” funcionando no Windows ou um programa que formata discos rígidos, certo? É certo que os programas descritos acima precisam acessar o hardware (o monitor no programa “Alô mundo”, o disco rígido no programa de formatação e a memória no programa que escreve na memória de outro programa), mas devido as restrições do Windows eles só podem fazer isso usando uma função do próprio Windows. Esses programas provavelmente usaram funções como WriteConsole (escrever na tela), SHFormatDrive (formatar) e WriteProcessMemory (escrever na memória). Além dessas funções há uma infinidade de outras funções que estão à disposição dos programadores, este conjunto de funções costuma receber o nome de API (Application Programming Interface) ou syscalls (chamadas do sistema). A utilidade de um Sistema Operacional e consequentemente a utilidade do Windows não é apenas restringir acessos ao hardware e evitar inconvenientes. Acontece que um programa que escreve na tela, como o “Alô mundo” mencionado nesta página, precisaria no mínimo de algumas dezenas de

linhas de código para fazer o serviço. No caso do programa que formata discos rígidos o aumento de linhas é consideravelmente maior, você pode considerar, sem exageros, alguns milhares linhas de código. Com a ajuda do Windows e sua API podemos escrever programas com muito mais rapidez.

Agora não é porque o Windows utiliza a função WriteConsole que outros sistemas deverão usar também. No Linux, por exemplo, o processo para escrever na tela é completamente diferente e jamais um programa escrito em Assembly para o Windows vai executar dentro do Linux sem nenhuma camada intermediaria que converta, por exemplo, a função WriteConsole do Windows para a função correspondente no Linux. Curiosamente um programa deste tipo realmente existe e é conhecido como wine. O wine converte as funções da API do Windows para uma função correspondente no Linux e o resultado acaba ficando muito semelhante. A dificuldade, entretanto, é desmotivadora, pois existe uma infinidade de funções na API do Windows e algumas, pra ajudar, não são documentadas. Tudo que um programa executando por intermédio do wine precisa fazer para "travar" é chamar uma função que o wine ainda não programou. De qualquer maneira, o wine vem fazendo um trabalho excelente.

Resumo

- O computador é uma máquina que permite a entrada de dados por algum dispositivo de entrada, efetua algum processamento nestes dados e transmite uma informação por algum dispositivo de saída.
- Este texto pretende explicar o funcionamento do computador de uma maneira razoavelmente detalhada dentro do contexto de um programador de baixo-nível.
- Infelizmente não vou trazer maiores detalhes sobre programação de Sistemas Operacionais, então só passarei o que pode ser feito por um programa dentro do Windows.
- O processador que vamos estudar será um processador do tipo x86 de 32 bits. Estes processadores podem ser produzidos por fabricantes diferentes e todos que recebem a designação x86 têm uma arquitetura compatível os demais x86.
- Um processador x86 não precisa necessariamente estar em um PC, mas este texto assume que o leitor está usando um computador PC. PC é geralmente associado a um computador baseado no IBM PC.
- Este texto assume que o leitor esteja usando um dos sabores do Windows (mas não exagere, tem que ser pelo menos um Windows 98.).
- O Sistema Operacional afeta um programa feito em assembly por que as funções que o programa chama em um Sistema A é diferente das funções de um Sistema B. O Sistema Operacional, tal como o Windows, não permite o acesso indiscriminado de um programa ao hardware.
- Assembly é uma linguagem na forma de mnemônicos – apelidos de fácil memorização – que representa a linguagem da máquina – processador. O processador dispõe de uma série de instruções ou comandos internos – "opcodes" ou código operacional – que representa basicamente o que este processador pode fazer.