Compiladores

e

processamento

Linguagem C

Verificar a existência dos pacotes:

- GCC (Gnu C Compiler);
- GDB (Gnu Debuger);
- Libc (blibliotecas C);
- nano, vi, emacs ou outro editor;
- make (caso não tenha: $ sudo apt-get install build-essential).

Linguagem C

Comandos:

$ dpkg --get-selections | grep gcc

$ dpkg --get-selections | grep gdb

$ dpkg --get-selections | grep libc

$ dpkg --get-selections | grep make

Linguagem C

Verificando versão instalada:

$ gcc --version

$ gdb --version

$ make --version

Linguagem C

GCC é um compilador completo da linguagem C ANSI (padrão internacional ISO), com suporte a Objective C, Java e Fortran.

A compilação em C envolve 4 fases/estágios:

- pré-processamento;
- compilação;
- assembly;
- e ligação (ou linkedição ou linker).

Linguagem C

O GCC é capaz de pré-processar e compilar vários arquivos com código-fonte em um ou mais arquivos em assembler (compilador Assembly).

O arquivo de assembler irá resultar em um ou mais arquivos objetos (obj) e estes serão ligados (linkeditados) às bibliotecas (linking) para enfim tornarem um arquivo executável.

O pré-processamento é responsável por expandir as macros e incluir os arquivos de cabeçalhos (.h) no arquivo-fonte.

O resultado é um arquivo que contém o código-fonte expandido.

Linguagem C

Etapas da compilação de um programa C

Código-fonte (.c): pré-processamento - expande as macros do código-fonte →

→ **Código pré-processado (.i):** compilação para o processador determinado →

→ transforma o código expandido em linguagem de máquina →

→ **Código Assembly (.s):** transforma o código de máquina em objeto →

→ **Objeto (.o):** realiza a ligação (linkedição) →

→ **Executável:** faz a ligação de objeto com as bibliotecas ...

Linguagem C

Exemplo:

O pré-processamento de um simples exemplo como o abaixo irá gera umarquivo C com mais de 900 linhas de código expandido:

Abra um terminal e digite:

Linguagem C

```
$ nano teste.c

#include<stdio.h>
int main()
{
printf(“Teste do compilador GCC ...!”);
}
```

Agora salve-o:

Ctrl + x

S

Após isso execute os comandos:

$ gcc -E teste.c -o teste.i

Linguagem C

Compare o tamanho dos arquivos .c e .i.:

$ ls -l teste*

-rw-rw-r-- 1 user user **55** May 15 15:43 teste.c

-rw-rw-r-- 1 user user **17522** May 15 15:43 teste.i

Obs:

A opção -E diz ao compilador para fazer o pré-processamento do código-fonte.

Linguagem C

O próximo passo da compilação será responsável por traduzir o código-fonte pré-processado em linguagem Assembly (linguagem menmônica) para um processsador específico:

```
$ gcc  -S  teste.i
```

Obs: o compilador tem a configuração do processador armazenada durante a sua instalação, logo, o código para o processador é realizado automáticamente.

Linguagem C

A seguir entra-se na fase do Assembler, em que o gcc converte o código de máquina de um processador específico em um arquivo-objeto.

Se no código existirem chamadas externas de funções, o gcc deixa seus endereços indefinidos para serem preenchidos posteriormente pelo estágio de ligação.

Linguagem C

O próximo comando irá fazer a compilação do código assembly do exemplo para o código de máquina:

```
$ as teste.s -o teste.o
```

Obs:

O comando as é um compilador assembler disponível no pacote GNU GCC.

Linguagem C

O arquivo teste.o possui instruções de máquina do código-exemplo com a referência indefinida para a função printf.

A linha onde consta “call printf” do código assembly indica que esta função está definida em uma biblioteca que deverá ser chamada durante o processamento.

Linguagem C

O último estágio é chamado de ligação (linker) e é responsável por ligar os arquivos objeto para criar um arquivo executável.

Ele faz isso preenchendo os endereços das funções indefinidas nos arquivos objeto com os endereços das bibliotecas externas do sistema operacional.

Isto é necessário porque os arquivos executáveis precisam de muitas funções externas do sistema e de bibliotecas para serem executadas.

Linguagem C

As bibliotecas podem ser ligadas ao executável preenchendo seus endereços nas chamadas externas de funções (dinâmica) ou de forma estática quando as funções das bibliotecas são copiadas para o executável.

No primeiro caso, o programa utilizará bibliotecas de forma compartilhada (sharing) e ficará dependente delas para funcionar.

Este esquema economiza recursos, pois uma biblioteca utilizada por muitos processos precisa ser carregada apenas uma vez na memória, logo, o tamanho do executável será pequeno, pois só terá os links para as bibliotecas.

No segundo caso, o programa será independente, uma vez que as funções e bibliotecas de que necessita estarão no código, logo este esquema permite que se houver mudanças de versão das bibliotecas, ele não seja afetado.

As desvantagens serão o tamanho do executável e as necessidades de mais recursos.

Linguagem C

Internamente a fase de ligação (linkedição ou linking) é bem complexa, mas o compilador faz isso de forma transparente para o usuário/programador através do comando:

```
$ gcc teste.o -o teste
```

Linguagem C

A seguir será gerado o arquivo “executável” chamado teste.

Para executá-lo basta utilizar o comando:

$ ./teste

Obs:

O ./ indica que o arquivo está no diretório local e o teste indica o nome do arquivo executável.

Este programa foi ligado às bibliotecas de forma compartilhada.

O comando ldd informa quais bibliotecas (dinâmicas) estão ligadas ao executável:

```
$  ldd  teste
```

Pode-se visualizar o tamanho dos arquivos gerados até então usando o comando:

```
$ ls -l teste*
```

Usando a opção -static copia e cola as funções/bibliotecas externas para o executável gerado acima:

```
$ gcc -static teste.c -o teste1
```

Linguagem C

Para realizar a verificação das bibliotecas (dinâmicas) ligadas ao executável executar o comando:

```
$ ldd teste1
```

Obs:

Com o comando a seguir pode-se comparar o tamanho dos executáveis gerados, o executável com funções/bibliotecas linkada e as funções/bibliotecas estáticas, verificando que a diferença é muito grande.

Linguagem C

Código objeto ou arquivo objeto é o nome dado ao código resultante da compilação do código fonte.

Para cada arquivo de código fonte é gerado um arquivo com código objeto, que posteriormente é "ligado" aos outros, através de um linker, resultando num arquivo executável ou biblioteca.

Um arquivo objeto não só contém código objeto, mas também informações sobre alocação de memória, os símbolos do programa (como nomes de variáveis e de funções) e também informações sobre debug.

Para visualizar o debuger pode-se usar a opção -g:

```
$  gcc  teste.c  -o  -g  teste
```

Linguagem C

Exercício

$ nano firstprog.c

```
#include <stdio.h>
int main()
{
	int i;
	for (i=0; i<10; i++)
	{
		puts(“Hello, Word!\n”);
	}
return 0;
}
```

Linguagem C

Gerar o arquivo a.out, que por padrão é o arquivo executável caso não tenha sido definido o nome de saída do arquivo executável:

```
$  gcc  firstprog.c
$  ./a.out
$  ls  -l  a.out
```

Assembly

Para verificar o código assembly podemos utilizar o utilitário objdump, que é utilizado para examinar os binários compilados.

A seguir o comando para examinar o código assembly referente ao main:

```
$ objdump -D a.out | grep -A20 main.:
```

Assembly

Há 2 principais padrões em Assembly:

- AT&T;
- e Intel.

Basicamente, os códigos mnemônicos assembly quando da presença do símbolo “%” indicam que este código está no padrão AT&T e os códigos sem o “%” indicam que estão no padrão Intel.

Assembly

Para visualizar o código mnemônico assembly no formato Intel deve-se utilizar a opção -M intel:

$ objdump -M intel -D a.out | grep -A20 main.:

Obs:

A opção -A20 indica que deve-se exibir as primeiras 20 linhas após a função main.

Assembly

O comando a seguir é utilizado para mostrar o estado dos registros do processador antes do programa começar:

```
$ gdb -q ./a.out
```

Assembly

```
$ gdb  -q  ./a.out
Reading symbols from ./a.out...(no debugging symbols found)...done.

(gdb) break main
Breakpoint 1 at 0x40052a

(gdb) run
Starting program: /home/lab/a.out
Breakpoint 1, 0x000000000040052a in main ()

(gdb) quit
A debugging session is active.
        Inferior 1 [process 4983] will be killed.

Quit anyway? (y or n) y
```

Java

Java é uma linguagem que foi criada para revolucionar a programação de aplicativos tendo com dois de suas principais características para isso:

- a escalabilidade: foi criado inicialmente para executar principalmente em sistemas embarcados, mas que atualmente opera em praticamente todas as plataforma de hardware;

- a portabilidade: indiferente da plataforma e do sistema operacional, tendo a Máquina Virtual Java, seu código pode ser executados por todos.

Java

Java tem dois processos de execução de código fonte.

Para executar algum código no Java precisa-se seguir dois passos:

- a interpretação;

- e a compilação.

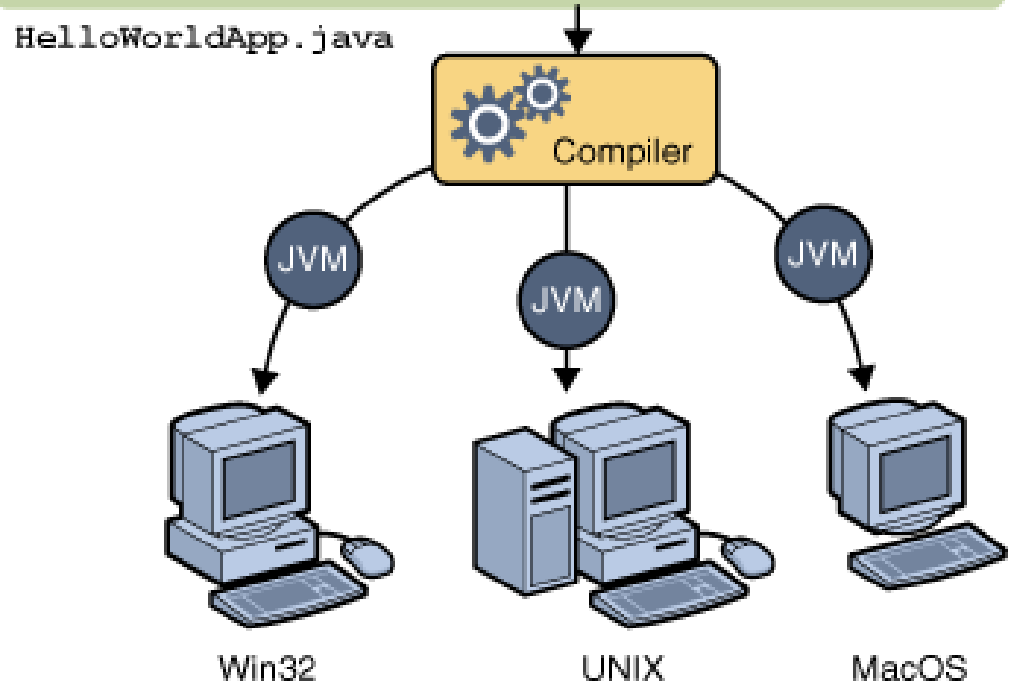
O usuário não precisa fazer muito, já que o próprio Java faz quase tudo.

Java

Java Program

```
class HelloWorldApp {
     public static void main(String[] args) {
          System.out.println("Hello World!");
     }
}
```

HelloWorldApp.java



Java

Compilação

Quando se cria uma classe Java, como o da figura anterior, e salva-se com a extensão .java, o código ainda Java não inteligível pela JVM.

Antes então que o hardware execute o código, precisa estar em linguagem de máquina, portanto, compilado!

Java

Compilação

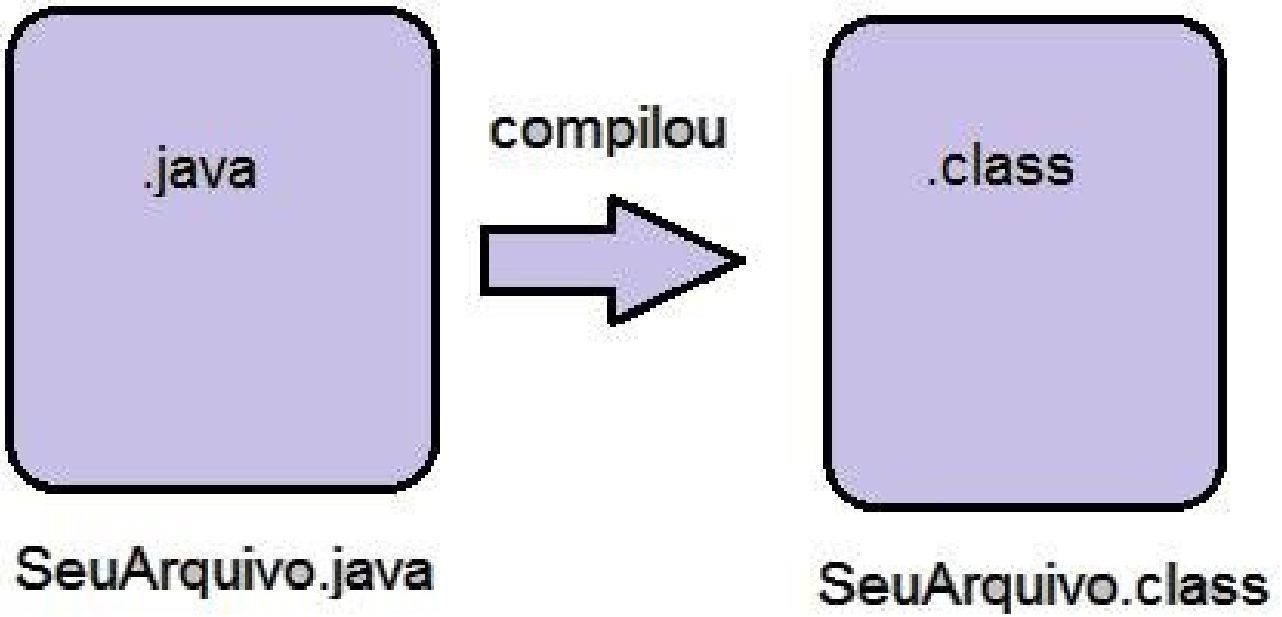
Assim, a JVM não consegue entender código Java, mas sim bytecodes.

Mas afinal, o que é Bytecode?

Bytecode é o resultado de um processo semelhante ao de compilação de código-fonte, mas que, no entanto, não pode ser executável.

Simplificando: é o código gerado quando você compila um arquivo .java

Java



Java

Compilação

Para compilar um arquivo Java, via linha de comando, usa-se o comando:

```
$  javac ola_mundo.java
```

Java

Compilação

Caso a classe Java não tiver nenhum erro, a classe será compilada e criará um novo arquivo:

ola_mundo.class

É nesse arquivo .class está o bytecode.

Java

Interpretação

O Java também precisa percorrer o código pra ver se todas as variáveis, endereçamentos e bibliotecas estão corretos.

Após esta verificação, o interpretaodr irá “traduzir” o código.

Assim, interpretar o código é fazer com que ele possa ser entendido pela máquina onde ele está sendo executado.

Java

Interpretação

O comando “java” (ou javaw) irá interpretar o arquivo .class e executar o código:

$ java ola_mundo

A saída esperada será:

Hello World!

Java

Assim, as fases, desde escrever o código, até a sua execução são (Deitel, 2014):

Fase 1 - Edição: o código-fonte é criado em um editor e armazenado em disco em um arquivo com extensão “.java”;

Fase 2 - Compilação: o Compilador cria o bytecode e armazena em disco em um arquivo com extensão “.class”;

Fase 3 Carregamento: o carregador de classe lê o arquivo “.class” que contêm bytecodes a partir do disco e coloca esses bytescodes na memoria.

Java

Fase 4 - Verificação: o verificador de bytecodes confirma que todos os bytes são validos e não violam restrições de segurança do java;

Fase 5 - Execução: para executar o programa , a “JVM” lê os bytecodes e os compila em “real-time” (tempo real) ou Just-in-time (JIT) para uma linguagem que o computador possa entender.

Java

Nas primeiras versoes do java, a JVM era simplesmente um interpretador para bytecodes java.

Isso fazia com que a maioria dos programas java executasse lentamente porque a JVM interpretava um bytecode por vez.

As JVMs atuais executam bytecodes utilizando uma combinação de interpretação e a chamada compilação Just-In-Time(JTI).

Java

A JVM analisa os bytecodes à medida que eles são interpretados.

Procura por “pontos ativos” (hot spots), partes dos bytecodes que executam com mais frequência.

Para essas um JIT traduz os bytecodes para a linguagem de máquina do hardware que está executando.

Java

Quando a JVM encontra novamente essas partes compiladas, o código de linguagem de maquina mais rapido é executado.

Assim, os programas java na realidade passam por duas fases de compilação:

- uma em que codigo-fonte é traduzido em bytecodes (para a portabilidade entre JVMs em diferentes plataformas de computador);
- e uma segunda: em que , durante a execução, os bytecodes são traduzidos em linguagem de máquina para o computador real em que o programa é executado.

Interpretação X Compilação

Interpretação

O interpretador analisa o código e executa a instrução (uma por uma) de forma simples e direta.

Ex:

Shell-script

Interpretação X Compilação

$ nano script.sh

```
echo -e "\n\n"
echo -e "\tOlá, seja bem vindo ...!\n"
echo -e "\tQual o seu nome? ..."
read nome
echo -e "\tPor favor, digite qualquer tecla ...\n"
eject
eject -t
sleep 3
echo -e "\tObrigado $nome ... volte sempre!\n"
```

Interpretação X Compilação

Compilação

O compilador analisa a instrução, traduz para a linguagem de máquina, e produz como saída um outro arquivo (executável) que a seguir será executada fazendo o que está descrito no código.

Ex:

Linguagem C, C++

Interpretação X Compilação

$ nano ola_mundo.c

```
#include <stdio.h>
int main()
{
	int i;
	for (i=0; i<10; i++)
	{
		puts("Hello World ...\n");
	}
return (0);
}
```

Interpretação X Compilação

Compilando:

$ gcc ola_mundo.c -o ola_mundo

Interpretação X Compilação

Executando:

$ ./ola_mundo

...