
Programando em Linux - Out2000
Ademar de Souza Reis Jr.
http://www.ademar.org
v1.3 - Out2000

Introdução

Este documento visa passar algumas explicações sobre a programação em ambiente linux, e é voltado para estudantes e programadores que queiram migrar de outros ambientes ou ter uma inicação à programação nesse SO. Vamos aqui abordar algumas características do sistema e alguns utilitários básicos.

Para a leitura deste, você deve ter um pequeno conhecimento prévio do sistema de arquivos do linux, sua estrutura de diretórios e usuários, além de uma certa familiaridade com a sintaxe de comandos. Você também deve saber consultar uma página manual e documentação info, e ainda assumimos que você domina um editor de textos simples, como o vi ou o emacs e, obviamente, têm conhecimentos de programação (os exemplos estão em C).

Se você ainda não se sente familiar com o sistema linux, é recomendado que, antes mesmo de continuar a leitura deste texto, você consulte outras fontes de documentação introdutória. Entre elas, estão dois livros em inglês (talvez existam traduções para o português) aqui listados, e lembre-se que há muita documentação de qualidade na internet que pode ser consultada gratuitamente. Os livros recomendados são:

- Running Linux 3rd edition
  Matt Welsh, Kalle Dalheimer, Lar Kaufman
  O'Reilly & Associates ISBN 156592469X

- The C Programming Language - 2th edition
  Brian W. Kernighan
  Prentice Hall ISBN 0131103628

Agora voltando ao nosso assunto principal, podemos dizer que o ato de programar (independente do ambiente, e do ponto de vista prático), consiste basicamente em:

escrever -> compilar -> depurar

Obviamente cada etapa pode ser um pouco extendida ou contraída, dependendo da complexidade de seu programa. (Um clássico "Hello World" talvez não precise de depuração, certo?).

Pra quem está acostumado a programar em outros ambientes, como em MS-DOS, essas etapas geralmene estão agrupadas em uma IDE (Interface de DEsenvolvimento), como o Borland C. Embora existam IDE's disponíveis para o ambiente do linux, a maioria dos programadores opta por trabalhar independentemente, escolhendo suas ferramentas favoritas.

Sendo assim, quando programamos em linux, geralmente utilizamos um utilitário para cada etapa. Isso apresenta algumas vantagens como flexibilidade e modularidade embora, a princípio, possa parecer um pouco mais complicado do que o usual. Com várias pequenas ferramentas, cada uma cumprindo seu papel, é possível chegar a grandes resultados... essa é uma das filosofias mais antigas e usadas no mundo UNIX, e não achamos que seja diferente na hora de

programar.

O que você precisa para criar um programa

Seguindo o raciocínio visto para as etapas da programação, precisamos dominar os seguintes tipos de utilitários:

1 - Editor de textos

Existem vários editores disponíveis no ambiente linux, sendo que alguns apresentam funcionalidades extras para programadores (como identação automática, sintaxe colorida, verificação de erros, etc).

Os dois editores mais utilizados são o vim (VI Improved) e o Emacs. Recomendamos que você utilize o editor de sua preferência, e deixamos a escolha para você. Simplesmente teste as alternativas e utilize aquele no qual você se sinta mais confortável. (Particularmente, prefiro o VI/vim. Embora seu aprendizado seja um pouco difícil ou até mesmo "traumático" para usuários iniciantes, ele se tornará seu "melhor amigo" na hora de programar).

2 - Compilador

A tarefa do compilador é a mais importante. Existem compiladores para as mais diversas linguagens no linux e, obviamente, você deve escolher aquele que melhor suprir às suas necessidades.

Os dois compiladores mais utilizados são o gcc e o g++. Respectivamente, GNU C Compiler e GNU C++ Compiler. (As coisas no mundo GNU mudam muito... na verdade, o gcc e o g++ são um único projeto, e, mais na verdade ainda, você pode compilar código C++ com o gcc e C com o g++, mas, no momento, não se preocupe com esses detalhes).

O mais importante para um programador conhecer em termos de compiladores são suas flags de compilação. Elas se dividem em:

- Warnings

Lhe ajudam a encontrar falhas na sintaxe e lógica do programa, falhas que muitas vezes não impedem a compilação deste. Principalmente se você programa em C, você deve utilizar várias flags, pois senão você corre o risco de que um código muito mal escrito seja compilado sem apresentar erros (o que extenderá a etapa de depuração).

- Otimização

Permitem que o compilador utilize instruções otimizadas ou específicas de um determinado processador, ou mesmo altere detalhes do código resultante, elminando redundâncias, "desenrolando" loops, etc. Existem também versões otimizadas dos principais compiladores para trabalhar com determinadas arquiteturas, como o Pentium, PentiumPro, etc.

3 - Depurador

Essa é a ferramenta que você utilizará quando as coisas não funcionam da primeira vez, ou em outras palavras, esse é a ferramenta que você utilizará sempre... :)

O depurador mais utilizado no ambiente linux é o gdb (GNU DeBuguer). Ele é um poderoso e excelente depurador para as linguagens C, C++ e Fortran. Existem inúmeros "frontends" gráficos para o GDB e um dos mais completos é o ddd (Data Display Debuger). Vale a pena você dar uma olhada se você prefere uma interface gráfica à console texto. Consulte a documentação apropriada (no caso, a página manual é um bom começo).

4 - Outros

Existem diversos outros utilitários que auxiliam muito o programador no ambiente linux. Um dos mais indispensáveis e úteis é o make, que permite otimizar e automatizar várias etapas da compilação e é indispensável quando trabalhamos com vários módulos. Ele será visto mais adiante.

Além deste, existem também bibliotecas para depuração, utilitários gráficos, ferramentas para layout, controle de versão, instalação, configuração, etc. Abaixo segue uma lista do que você pode achar interessante: (procure documentação a respeito nas páginas manuais e nos repositórios de software linux)

EletricFence
checker
indent
diff
patch
autoconf
cvs

De um modo geral, pelo que vimos até aqui você precisará dominar pelo menos quatro ferramentas para programar "satisfatóriamente" em linux: editor de textos, compilador, depurador e o make. Mas novamente eu insito: não pense que o número de ferramentas faz do linux um ambiente difícil para programação. O tempo gasto com o aprendizado valerá a pena no final.

Vamos agora analisar de uma forma mais detalhada os utilitários mais importantes, ou pelo menos indispensáveis, para programação em linguagem C:

gcc - O Compilador

Sem dúvida o compilador mais utilizado e importante dos sistemas Unix. Embora existam outros compiladores para a linguagem C disponíveis, o gcc é, de longe, o mais famoso e completo. Ele foi criado e mantido pela comunidade GNU, e está em estágio de ser considerado um dos melhores compiladores existentes para a linguagem C. Existem inúmeros detalhes que você não precisa saber no momento sobre o projeto do gcc, como codinomes, versões otimizadas, versões em desenvolvimento, etc... Simplesmente o chame de "gcc" e viva (ou melhor, programe) feliz.

A utilização é simples: após criar um arquivo com o código corretamente escrito, você deve invocar o gcc da seguinte forma:

```
$ gcc fonte.c
```

O resultado é a criação de um arquivo executavel de nome "a.out" (esse é o nome dado a novos executáveis por razões históricas, que não vem ao caso agora).

Se você quer que o gcc crie um arquivo executável com um nome específico (e geralmente você quer), você deve utilizar a opção (que chamaremos de "flag") "-o". Ela significa "outupt" (saída), e é utilizada antes do nome do arquivo que será criado.

Abaixo refaremos a compilação do exemplo anterior, mas criando um arquivo de nome "executavel":

```
$ gcc fonte.c -o executavel
```

Como você deve estar ciente, em linux não estamos presos à extensões de nome de arquivo (.exe, .com, .bin, etc) e, portanto, não há necessidade de colocarmos uma em nosso programa recém criado.

Para compilar programas que estejam em vários arquivos, você deve primeiramente compilar os módulos como objetos para então compilar o programa principal ou invocar o gcc com todos os arquivos de fonte ao mesmo tempo.

Vamos ver como exemplo o caso da compilação de um programa que esteja separado em três partes (arquivos):

```
interface.c
processamento.c
principal.c
```

A sequencia para a criação de nosso programa, que chamaremos de "teste" seria:

```
$ gcc interface.c processamento.c interface.c -o teste
```

ou

```
$ gcc -c inteface.c -o interface.o
$ gcc -c processamento.c -o processamento.o
$ gcc interface.o processamento.o principal.c -o teste
```

A flag "-o" já é por nós conhecida. A novidade fica por conta da flag "-c" (compilar mas não linkar) que é utilizada para a criação do arquivo objeto.

O processo de gerar os objetos para então criar o executável se torna cansativo e demais trabalhoso para ser feito sem uma automação. Para essa tarefa veremos o utilitário "make" logo abaixo.

Outras flags bastante importantes são as de "warnings" (alertas). São elas que vão detectar problemas em nosso código que vão desde falta de estilo até problemas com passagem de parâmetros e, principalmente, ponteiros e estruturas complexas.

As mais utilizadas e por mim recomendadas são:

```
-pedantic -Wall -W -Wtraditional -Wshadow -Wpointer-arith
```

```
-Wbad-function-cast -Wcast-qual -Wcast-align -Wwrite-strings -Wconversion
-Waggregate-return -Wmissing-prototypes -Wmissing-declarations
-Wnested-externs -Winline -Wwrite-strings
```

A explicação de cada uma dessas opções é muito extensa para ser explicada aqui. Para isso é altamente recomendado que você faça uma consulta à documentação do gcc (páginas manuais e info).

Note que algumas dessas opções podem se tornar incômodas na criação de programas que, por um motivo ou outro, precisam quebrar certas "regras" para chegar a algum resultado ou tenham um grau de complexidade alto.

gdb - Depurador

Existem vários depuradores disponíveis para o ambiente do linux, mas o mais utilizado e suportado com certeza é o gdb. Como o gcc, ele foi criado e é mantido pela comunidade GNU, e apresentando um ambiente simples mas recursos bastante avançados, é considerado um dos melhores depuradores existentes para o ambiente Unix (sendo então usado por muitos "frontends").

Nosso objetivo é conferir a utilização básica do gdb. Uma documentação mais completa você encontra nas páginas manais e nos docuemtos info. Os roteiros aqui passados são válidos para praticamente qualquer "frontend" do gdb. Lembre-se: Um frontend não é nada mais nada menos que uma "carinha mais bonita" para seu verdadeiro programa. Vale a pena entender o que acontece de verdade.

Executar o gdb é simples. Primeiramente, você deve compilar seu programa (e módulos objeto) com a flag -g ou uma das equivalentes, que incluirá no programa executável informações necessárias à depuração. Já ao executar o gdb, é bom que você tenha acesso ao código fonte de seu programa para ir conferindo a execução. ("frontends" gráficos geralmente fazem isso por você também).

Você deve invocar o gdb como abaixo:

```
$ gdb arquivo_executável
```

E sua interface é bastante simples:

```
GNU gdb 4.17.0.12 with Linux support
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-Debian/GNU-linux"...
(gdb)
```

Agora você só precisa utilizar os comandos necessários para acompanhar a execução do código, depurando-o, utilizando a técnica básica:

- Insira um breakpoint no ponto onde você deseja iniciar a depuração (linha N do código);
- Execute o programa (dentro do gdb);
- Avance na execução, conferindo os parâmetros passados às funções e o

tipo e conteúdo das variáveis, até encontrar seu erro.

A sintaxe dos comandos é simples: você tem o comando por extenso e o comando simplificado (geralmente a primeira letra do mesmo). Os parâmetros (como números de linha e nomes de variáveis) são passados imediatamente após o comando.

Os mais importantes são:

Inserir breakpoint na linha N:
break N
b N

Remover todos os breakpoints:
delete
d

Rodar o programa até o primeiro breakpoint ou fim do mesmo:
run <parametros>
r <parametros>
onde <parametros> sao os argumentos passados ao programa (como uma flag ou nome de arquivo)

Executar proximas N linhas:
next N
n N

Executar proximas N passos:
step N
s N

Matar (encerrar) o programa:
kill
k

Exibir informacoes sobre o tipo de uma variavel:
what <nome_var>
w <nome_var>

Exibir o conteúdo de uma variavel:
print <nome_var>
p <nome_var>

Visualizar pilha de recursao:
backtrace
bt

Obter ajuda em relação a algum comando:
help <comando>
h <comando>

Sair do gdb:
quit
q

A sintaxe em relação às variáveis é a utilizada na linguagem C, e o último comando pode ser repetido pressionando-se [ENTER].

Vamos agora conferir um exemplo bem simples em linguagem C:
(se você tiver dificuldades em entender o código, então você precisa estudar um pouco mais de C) :)

O programa a seguir tem como único objetivo o aprendizado do gdb. Ele não tem nenhuma utilidade, sentido ou objetivo lógico. Em futuras versões desse documento, exemplos mais úteis e didáticos serão incluídos.

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

#define MAGICO 16

void funcao1(char **c);
int funcao2(int x);

int main (void)
{
    int a = 100;
    char b;
    char *c;

    c = (char *) malloc(sizeof(char) * (MAGICO));

    funcao1(&c);
    a = funcao2(a);

    printf("%s\n", c);

    exit(0);
}

void funcao1(char **c)
{
    int i;

    for (i = 0; i < MAGICO; i++)
        *c[i] = 'a';
}

int funcao2(int x)
{
    return x * MAGICO + 50;
}
---------------------------------------------------------------------------

$ gcc exemplo.c -o exemplo -g
$ ./exemplo
Segmentation fault (core dumped)
```

A mensagem acima nos diz que o programa foi interrompido pelo kernel por tentar acessar uma área inválida ou não acessível da memória (Falha de Segmentação). Em nosso sistema, quando um processo é interrompido devido a um "Segmentation fault", um arquivo de "core" é criado. Esse arquivo contém informações úteis a respeito da memória no momento em que o processo estava sendo executado, e pode agora ser utilizado pelo gdb (ou outro depurador) para descobrir onde ocorreu o erro. Como programador, mantenha

seu sistema configurado para gerar arquivos de core (embora isso, geralmente, seja indesejável para usuários normais).

```
$ gdb exemplo core
GNU gdb 4.17.0.12 with Linux support
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-Debian GNU/Linux"...
Core was generated by `./exemplo'.
Program terminated with signal 11, Segmentation fault.
#0  0x80484b0 in funcao1 (c=0xbffff3bc) at exemplo.c:30
30                      *c[i] = 'a';
(gdb) print i
$1 = 4
(gdb)
```

Como você pode notar, o gdb, a partir do arquivo de core gerado juntamento com o segmentation fault, já detecta em que linha ocorreu o erro, e você pode executar comandos como "print i". Isso muitas vezes já é o suficiente para detectar o erro. Mas vamos executar o programa passo a passo assim mesmo:

```
(gdb) b 10
Breakpoint 1 at 0x8048430: file exemplo.c, line 10.
(gdb) r
Starting program: /home/ademar/prog/exemplo


Breakpoint 1, main () at exemplo.c:10
10      {
(gdb) next
main () at exemplo.c:11
11              int a = 100;
(gdb) next
15              c = (char *) malloc(sizeof(char) * (MAGICO + 1));
(gdb) next
17              funcao1(&c);
(gdb) step
funcao1 (c=0xbffff39c) at exemplo.c:29
29              for (i = 0; i < MAGICO; i++)
(gdb) print *c
$2 = 0x8049640 ""
(gdb) next
30                      *c[i] = 'a';
(gdb) print i
$4 = 0
(gdb) next 8
30                      *c[i] = 'a';
(gdb) print i
$15 = 4
(gdb) print *c
$18 = 0x8049640 "a"
```

Ops. Tem coisa errada aqui. O comando print seguido de uma variável que seja um vetor de caracteres (string), imprime todo o seu conteúdo, desde o

primeiro caracter até o '\0'. (como você verá mais adiante ao executarmos a versão corrigida ainda no gdb). O que vemos aqui é que o vetor tem apenas um caracter.

Pensando um pouco, conseguimos notar que estamos acessando o vetor de maneira errada. O correto seria (*c)[i], e não *c[i].

Problema resolvido. :)

Agora vamos executar o programa (corrigido) até seu término dentro do gdb, utilizando os comandos simplificados (apenas a primeira letra):

```
$ gcc exemplo.c -o exemplo -g
$ ./exemplo
aaaaaaaaaaaaaaa
```

A linha acima é o resultado de nosso programa em execução. Útil, não? :)

```
$ gdb exemplo
[...]
(gdb) b 10
Breakpoint 1 at 0x8048430: file exemplo.c, line 10.
(gdb) r
Starting program: /home/ademar/prog/exemplo

Breakpoint 1, main () at exemplo.c:10
10      {
(gdb) n
main () at exemplo.c:11
11              int a = 100;
(gdb) n
15              c = (char *) malloc(sizeof(char) * (MAGICO + 1));
(gdb) n
17              funcao1(&c);
(gdb) s
funcao1 (c=0xbffff39c) at exemplo.c:29
29              for (i = 0; i < MAGICO; i++)
(gdb) n
30                      (*c)[i] = 'a';
(gdb) n 30
30                      (*c)[i] = 'a';
(gdb) p i
$1 = 15
(gdb) p *c
$2 = 0x8049640 'a' <repeats 15 times>
```

Agora sim. O vetor "c" contém 15 vezes a letra 'a'.

```
(gdb) n
29              for (i = 0; i < MAGICO; i++)
(gdb) n
31      }
(gdb) n
main () at exemplo.c:18
18              a = funcao2(a);
(gdb) p a
$3 = 100
(gdb) n
```

```
20              printf("%s\n", c);
(gdb) p a
$4 = 1650
(gdb) n
aaaaaaaaaaaaaaa
22              exit(0);
(gdb) n

Program exited normally.
(gdb)
```

Esse é o fim da execução.

Este exemplo é bastante simples, mas se você teve dificuldades em acompanhar, crie um programa ainda mais simples (algo como um "hello world") e faça os testes. Você verá grande utilidade no gdb principalmente quando estiver trabalhando com estruturas e ponteiros, pois com variáveis desse tipo é praticamente impossível a depuração usando-se "printf's", e o comando "print" do gdb lhe surpreenderá!

make

Basicamente, o make é um "automatizador" de comandos. Ele pode ser utilizado para agrupar um conjunto de comandos qualquer em um "rótulo" que chamamos logo após o comando "make", e é extremamente útil quando estamos trabalhando em um projeto relativamente grande (com vários módulos a compilar) ou quando usamos várias opções (flags) de compilação - o que, como vimos acima, é indispensável.

Para utilizá-lo, você deve criar um arquivo com os rótulos e os comandos que ele executará. Esse arquivo deve chamar-se Makefile ou makefile (o nome é indiferente, pois o formato e utilização dos arquivos é a mesma, e quando precisar-mos citá-los, vamos citar apenas "Makefile").

O formato de um Makefile simples é o seguinte:

```
# comentário
rótulo: dependências
<TABULACAO>comando1; \
           comando2; \
                ...; \
           comandon;
```

Onde:

comentário
    É um texto ignorado, serve para documentar o arquivo.

rótulo
    Em compilações, deve ser o nome do arquivo resultante da utilização dos comandos (geralmente resultado da compilação).

dependências
    Nome dos arquivos e/ou rótulos que devem ser verificados antes da execução dos comandos (veja explicação abaixo)

```
<TABULACAO>
    É uma tabulação comum. Não deve conter espaços, apenas uma tabulação.
    O principal erro ao criar-se um arquivo Makefile reside aqui...

comandos
    A lista de comandos que devem ser executados pelo make para o respectivo
    rótulo. Ao final de cada linha, deve-se inserir uma barra inversa "\",
    para continuar a lista (ou até mesmo um comando único) na linha logo
    abaixo.
```

É comum especificarmos algumas "variáveis" locais ao arquivo do make para facilitar a sintaxe, deixando-a mais simples. A utilização de variáveis é similar à maneira como usamos na shell bash, como mostrado abaixo:

```
Atribuição:
    VARIAVEL=valor
Utilização:
    $(VARIAVEL)
```

Uma das grandes vantagens em se utilizar o make é a verificação das dependências. O funcionamento é o seguinte: antes de executar os comandos de um determinado rótulo, o make verifica se os rótulos listados na lista de dependências já foram executados. Para isso, ele verifica se o arquivo rótulo (que, como citado, deve ser o nome do arquivo resultante da compilação) é mais recente que suas dependências. Esta é uma forma de evitar a recompilação de módulos que já estejam atualizados, e sempre deve ser utilizada, pois otimiza em muito esse processo (tempo de compilação).

Abaixo temos um exemplo de arquivo Makefile comentado, que tem como objetivo criar um programa de nome teste, utilizando uma série de flags e opções para o gcc (através da utilização de variáveis).

```
# Compilador usado (C Compiler)
CC=gcc

# Opcoes diversas para o gcc
# Para uma explicação detalhada sobre a utilidade de cada opção, veja a
# documentação do gcc (man / info)

OVERALL_OPTIONS=-pipe

DIALECT_OPTIONS=-ansi

# Destaque para as opções de warning abaixo. Elas podem ser extremamente úteis
# na detecção de erros, mas algumas podem também impedir (ou pelo menos
# atrapalhar) a compilação de programas complexos.
WARN_OPTIONS=-pedantic -Wall -W -Wtraditional -Wshadow -Wpointer-arith\
         -Wbad-function-cast -Wcast-qual -Wcast-align -Wwrite-strings\
         -Wconversion -Waggregate-return\
         -Wmissing-prototypes -Wmissing-declarations\
         -Wnested-externs -Winline -Wwrite-strings


DEBUG_OPTIONS=-g

LDFLAGS=-lm

# une todas as opções de compilação em uma variavel só (CFLAGS)
```

```
CFLAGS=$(OVERALL_OPTIONS) $(DIALECT_OPTIONS) $(WARN_OPTIONS) $(DEBUG_OPTIONS)\
       $(CODE_GENERATION_OPTIONS)

# all - usado pra simplificar, permitindo o uso do comando make sem nenhum
# parâmetro (rótulo). Na verdade, o nome "all" é apenas sugestivo... poderia
# ser qualquer outro. O detalhe é que, ao invocarmos o make sem nenhum
# parâmetro, ele entrará no primeiro rótulo.
all: teste

# programa principal
teste: modulo.o teste.c
    $CC teste.c modulo.o -o teste $(CFLAGS)

# compilacao de um modulo
modulo.o: modulo.c
    $CC -c modulo.c -o modulo.o $(CFLAGS)

# excluir tudo o que foi compilado
clean:
    rm *.o teste -f
```

Como sempre, vale a pena conferir a documentação relativa ao make. Ele é muito mais poderoso do que você pode imaginar...

Conclusão

O linux foi criado por programadores e para programadores (embora hoje estejamos em uma era de "usuário final"). Programar nesse ambiente exige um pouco de conhecimento e aprendizado, mas o resultado vale a pena principalmente se você pretende realmente entender o que está acontecendo a cada passo (ou seja, quer programar "de verdade").

Juntamente com outros SO's de caráter livre, o linux se destaca por apresentar seu código (e da maioria dos programas) aberto e centenas de novos softwares (simples ou complexos, úteis ou não tão úteis assim) são lançados diariamente (acompanhe grandes sites como o FreshMeat (www.freshmeat.net). Aprender a programar e, em especial, ganhar experiência, é muito mais simples em um ambiente como este.

Domine os utilitários e entre para uma equipe de um pequeno projeto... em pouco tempo, você estará dando sua contribuição para a comunidade de software livre e em menos tempo ainda, já estará escrevendo seus próprios programas e lançando-os para ajudar a outros.

Já ia me esquecendo:
Bem vindo ao maravilhoso mundo da programação sem fronteiras...

Futuras atualizações

- Melhores exemplos
- diff, patch, autoconf, automake, etc.

Redistribuição e Copyright

Esse documento foi escrito por Ademar de Souza Reis Junior

<ademar@ademar.org>.

Contribuições (como melhores exemplos e explicação de outros utilitários)
são muito bem vindas.

Copyright (c) 2000 Ademar de Souza Reis Jr. <ademar@ademar.org>
Redistribuição permitida e incentivada.