Desenvolvimento em UNIX

No processo de codificação de um software, são atividades básicas:

- Edição
- Compilação
- Execução
- Depuração

Além destas, também são atividades necessárias, sobretudo à medida em que os softwares desenvolvidos crescem em tamanho e complexidade:

- Construção de bibliotecas
- Gerência de versões
- Documentação

Neste módulo serão enumeradas algumas ferramentas de código aberto (*open source*) disponíveis para auxiliar nas diversas etapas do desenvolvimento de programas C/C++ em plataformas UNIX, mais particularmente no mundo Linux.

Edição de código

Para a edição de código-fonte pode-se optar por editores de código-fonte como Emacs [http://www.gnu.org/software/emacs/emacs.html] (ou XEmacs [http://www.xemacs.org]), Jed [http://www.jedsoft.org/jed/], Vi [http://www.vim.org/], Nano [http://www.nano-editor.org/], Nedit [http://www.nedit.org/], Kate [http://kate.kde.org/] (no ambiente KDE), Gedit [http://gedit.sourceforge.net/] (no ambiente Gnome) e outros. Pode-se também optar por ambientes de desenvolvimento integrado (IDEs) como Eclipse [http://www.eclipse.org/], Kdeveloper [http://www.kdevelop.org/], Anjuta [http://anjuta.sourceforge.net/] e inúmeros outros.

Qual seria a melhor opção ? Esta é uma pergunta difícil de responder, pois há muitas controvérsias...

Compilação

Existem vários compiladores C/C++ disponíveis livremente em ambiente Linux, mas o GCC [http://gcc.gnu.org/] (GNU Compiler Collection) é de longe o compilador mais popular e certamente um dos melhores. O GCC compreende compiladores de front-end e geradores de código para várias linguagens de programação (C, C++, ObjC, Fortran, Ada, Java, ...) em diversas plataformas [http://gcc.gnu.org/install/specific.html].

O GCC é um compilador que opera em linha de comando. Por default, o compilador irá realizar a compilação e ligação do(s) fonte(s), gerando um arquivo executável denominado `a.out` (que tem esse nome por razões históricas). Para executar esse arquivo basta invocá-lo, em uma linha de comando:

```
// arquivo hello.c
int main ()
{
  printf ("Hello, world\n") ;
```

```
}

~> gcc hello.c

~> ls -l
-rwxr-xr-x 1 prof 11724 Set 28 16:25 a.out
-rw-r--r-- 1 prof 45 Set 28 16:13 hello.c

~> a.out
Hello, world
```

Pode-se redefinir o arquivo de saída com a opção `-o`:

```
~> cc -o hello hello.c
```

É possível compilar diversos arquivos interdependentes simultaneamente, como mostra o seguinte exemplo:

```
// arquivo hello.c
int main ()
{
  escreva ("Hello, world\n") ;
}

// arquivo escreva.c
void escreva (char *msg)
{
  printf ("%s", msg) ;
}

~> cc -o hello hello.c escreva.c
```

Também é possível efetuar somente a compilação dos códigos-fonte, sem efetuar a ligação e a geração do executável. Nesse caso, serão gerados os arquivos com o código-objeto (`*.o`) dos correspondentes arquivos de código fonte (`*.c`), como mostra o exemplo abaixo:

```
~> cc -c hello.c escreva.c

~> ls -l
-rw-r--r--  1  prof     50  Set 28 16:13  escreva.c
-rw-r--r--  1  prof    788  Set 28 16:16  escreva.o
-rw-r--r--  1  prof     45  Set 28 16:13  hello.c
-rw-r--r--  1  prof    800  Set 28 16:16  hello.o
```

Esses arquivos objeto poderão posteriormente ser ligados, gerando o executável:

```
~> cc -o hello *.o

~> ls -l
-rw-r--r--  1  prof     50  Set 28 16:13  escreva.c
-rw-r--r--  1  prof    788  Set 28 16:16  escreva.o
-rwxr-xr-x  1  prof  11724  Set 28 16:25  hello
-rw-r--r--  1  prof     45  Set 28 16:13  hello.c
-rw-r--r--  1  prof    800  Set 28 16:16  hello.o
```

As opções usuais de execução do compilador GCC podem ser consultadas em sua página de manual (`man gcc`).

Usando bibliotecas

A linguagem C é rica em poder de expressão, mas relativamente pobre em funcionalidades. Para construir aplicações que fazem uso de funcionalidades específicas, como interfaces gráficas, comunicação via rede, fórmulas matemáticas complexas, etc, devem ser usadas bibliotecas. Algumas bibliotecas encapsulam chamadas do sistema operacional, sendo então chamadas de *bibliotecas de sistema*, enquanto outras provêem funcionalidades no espaço de usuário, como funções matemáticas e interfaces gráficas.

As bibliotecas mais comuns, utilizadas por todas as aplicações e utilitários do sistema, são:

- `libc`: na verdade um grande pacote de bibliotecas que provê funcionalidades básicas de entrada/saída, de acesso a serviços do sistema, à rede, etc.
- `ld-linux`: provê as funções necessárias para a carga de bibliotecas dinâmicas, durante a inicialização do programa.

Por default, essas duas bibliotecas são automaticamente incluidas e ligadas em todos os programas. Para compilar um programa que utiliza outras bibliotecas externas, algumas opções devem ser informadas ao compilador. Por exemplo, considere o seguinte programa, que faz o cálculo de Pi através de uma série de Gregory:

```
// arquivo pi.c
#include <math.h>
int main ()
{
  int i ;
  double pi = 0 ;

  for (i=0; i < 1000000; i++)
    pi += pow (-1,i) / (2*i+1) ;
  pi *= 4 ;

  printf ("O valor aproximado de Pi é: %f\n", pi) ;
}
```

Ao compilar esse programa, obtemos:

```
~> cc pi.c -o pi
/tmp/ccCANYTf.o(.text+0x42): In function `main':
: undefined reference to `pow'
collect2: ld returned 1 exit status
```

Esse erro ocorre porque o ligador não conseguiu encontrar a função `pow` em nenhum dos arquivos fonte ou objeto informados no comando, nem nas bibliotecas padrão. Essa função se encontra na biblioteca matemática `/usr/lib/libm`, que deve ser informada ao ligador da seguinte forma:

```
~> cc pi.c -o pi -lm
```

A opção `-lm` indica o uso da biblioteca `libm`. Da mesma forma, `-lpthread` indica o uso da biblioteca `libpthread`, e assim por diante. O ligador irá procurar por arquivos de bibliotecas nos diretórios padrão (`/lib`, `/usr/lib`, ...). Pode-se informar outros diretórios de bibliotecas ao ligador através da opção `-L`

(detalhada mais adiante neste texto).

Há duas formas básicas de ligar as bibliotecas a um programa executável:

- Na ligação estática (*static linking*), o código da biblioteca é incorporado ao executável. O executável fica maior, mas não depende de bibliotecas instaladas no sistema para poder executar.
- Na ligação dinâmica (*dynamic linking*), o executável guarda apenas referências às bibliotecas necessárias, que são resolvidas somente quando o programa executável for lançado. O executável fica muito menor, mas precisa que todas as bibliotecas necessárias estejam presentes no sistema para executar.

A ligação dinâmica é feita por default. Para compilar um programa, ligando-o estaticamente à bibliotecas, devemos executar:

```
~> cc -static pi.c -o pi -lm
```

O utilitário `ldd` permite verificar de quais bibliotecas dinâmicas um executável depende:

```
~> ldd pi
    libm.so.6 => /lib/i686/libm.so.6 (0x40028000)
    libc.so.6 => /lib/i686/libc.so.6 (0x4004b000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Para encontrar as bibliotecas dinâmicas, são percorridos os diretórios indicados pelo arquivo de configuração `/etc/ld.so.conf` e pela variável de ambiente `LD_LIBRARY_PATH`. O utilitário `ldconfig` permite atualizar as informações sobre bibliotecas dinâmicas nos diretórios padrão (ou nos diretórios informados via linha de comando).

Construindo bibliotecas estáticas

Para construir uma biblioteca de ligação estática são necessários vários passos. Inicialmente, todos os arquivos-fonte que irão compor a biblioteca devem ser compilados, para gerar seus arquivos-objeto correspondentes:

```
~> gcc -c file1.c
~> gcc -c file2.c
~> gcc -c file3.c
...
```

A seguir, deve ser usado o utilitário `ar` (*archiver*) para juntar todos os arquivos-objeto em uma biblioteca:

```
~> ar rvs libtest.a file1.o file2.o file3.o ...
```

Os flags `rvs` indicam:

- `r` (*replace*): substituir versões anteriores dos arquivos na biblioteca, caso existam
- `v` (*verbose*): mostrar na tela as inclusões que estão sendo realizadas
- `s` (*symbols*): criar uma tabela dos símbolos que estão sendo agregados à biblioteca

O utilitário `ar` possui diversos outros flags. Por exemplo, pode-se consultar o conteúdo de uma biblioteca estática:

```
~> ar t libtest.a
file1.o
file2.o
file3.o
```

Pode-se consultar todos os símbolos definidos em uma biblioteca estática (ou em qualquer arquivo objeto) através do utilitário `nm`:

```
~> nm libtest.a
file1.o:
00000000 T a
         U printf

file2.o:
00000000 T b
         U printf

file3.o:
00000000 T c
           U printf
```

A opção `-L.` é necessária para incluir o diretório corrente nos caminhos de busca de bibliotecas do ligador. Esta abordagem é melhor que a anterior, pois neste caso o ligador somente irá incluir no executável final os objetos que forem efetivamente necessários.

Observe que a biblioteca foi informada ao ligador na opção `-ltest`. Por default, ao encontrar uma opção `-labc`, o ligador irá procurar pela biblioteca `libabc.a` nos diretórios default de bibliotecas (`/lib`, `/usr/lib`, `/usr/local/lib`, ...) e nos diretórios informados pela opção `-L`.

Para atualizar/incluir qualquer arquivo da biblioteca, basta executar `ar` novamente, indicando o(s) arquivo(s) a atualizar/incluir:

```
~> ar rvs libtest.a file2.o
```

A forma mais simples de usar a biblioteca é indicá-la ao compilador no momento da ligação:

```
~> gcc -o meuprograma meuprograma.c libtest.a
```

Uma opção abreviada de ligação pode ser utilizada. Nela, não é necessário indicar o nome completo da biblioteca:

```
~> gcc -o meuprograma meuprograma.c -L. -ltest
```

Construindo bibliotecas dinâmicas

A construção de uma biblioteca de ligação dinâmica é um pouco mais complexa. Primeiro, é necessário compilar os arquivos-fonte que irão compor a biblioteca usando a opção `-fPIC`, que irá gerar código independente de posição (*PIC - Position Independent Code*):

```
~> gcc -fPIC -c file1.c
~> gcc -fPIC -c file2.c
~> gcc -fPIC -c file3.c
...
```

Depois, pode-se criar a biblioteca dinâmica, a partir dos arquivos-objeto:

```
~> gcc -g -shared -Wl,-soname,libtest.so.0 -o libtest.so.0.0 file1.o file2.o file3.o
```

Observe que a opção `-Wl` permite informar a opção `-soname=libtest.so.0` ao ligador. Essa opção permite definir o nome e versão da biblioteca.

Finalmente, para instalar a biblioteca, deve-se movê-la para o diretório adequado e gerar os links necessários:

```
~> mv libtest.so.0.0 /usr/local/lib
~> ln -s libtest.so.0.0 libtest.so.0
~> ln -s libtest.so.0 libtest.so

~> ls -l /usr/local/lib
lrwxrwxrwx  1  prof      12   Out 2 18:20  libtest.so -> libtest.so.0
lrwxrwxrwx  1  prof      14   Out 2 18:06  libtest.so.0 -> libtest.so.0.0
-rwxr-xr-x  1  prof    6914   Out 2 18:06  libtest.so.0.0
```

Para usar a biblioteca:

```
~> gcc -o meuprograma -L. -ltest meuprograma.c
~> meuprograma
```

Caso a biblioteca esteja em um diretório não listado em `/etc/ld.so.conf` (arquivo de configuração do carregador e ligador dinâmico), deve-se incluir o diretório nesse arquivo e a seguir executar `ldconfig`, ou informar o carregador dinâmico através da variável de ambiente `LD_LIBRARY_PATH`:

```
~> export LD_LIBRARY_PATH=.
~> meuprograma"
```

Organizando seu código

À medida em que um software cresce em tamanho e funcionalidades, seu código-fonte deve ser organizado corretamente para facilitar sua compreensão, manutenção e evolução. É importante quebrar o código-fonte em arquivos separados, dividindo-o de acordo com os módulos e/ou funcionalidades do sistema.

Ao dividir o código-fonte em arquivos separados, alguns cuidados devem ser tomados:

- Agrupe funções correlatas em um mesmo arquivo
- Coloque as assinaturas das funções públicas e as estruturas de dados necessárias às mesmas em um arquivo de cabeçalho `.h`, a ser incluso pelos demais arquivos que usarem essas funções.
- Somente efetue inclusões (`#include`) de arquivos de cabeçalho (`.h`). Evite inclusões de arquivos `.c` (deixe para o ligador efetuar a inclusão desse código no executável final).

O arquivo principal (`main.c`) deve incluir todos os arquivos de cabeçalho necessários e também deve definir a função `main`.

```
/* arquivo main.c */

#include "complex.h"

int main()
{
  complex_t a, b, c ;

  complex_define (&a, 10, 17) ;
  complex_define (&b, -2, 4) ;
  complex_sum (&c, a, b);
  ...
}
```

O arquivo de cabeçalho `complex.h` deve declarar somente informações públicas: tipos de dados e assinaturas de funções:

```
/* arquivo complex.h */

#ifndef __COMPLEX__
#define __COMPLEX__

typedef struct {
  float r,i;
} complex_t ;

void complex_define (complex_t *v, float r, float i) ;
...

#endif
```

Deve-se observar o uso das macros de pré-compilação `#ifdef` e `#define`, para evitar a repetição das definições no caso de múltiplas inclusões do arquivo de cabeçalho. O arquivo correpondente `complex.c` conterá então as informações privadas dos módulos: estruturas de dados internas, variáveis globais e implementações das funções.

```
/* arquivo complex.c */

#include "complex.h

void complex_define (complex_t *v, float r, float i)
{
  /* implementation of the function */
  ...
}

...
```

O arquivo `complex.c` pode ser compilado separadamente (gerando um arquivo objeto `complex.o` que poderá ser ligado ao arquivo `main.o` posteriormente). Essa organização torna mais simples a construção de bibliotecas e a distribuição de código binário para incorporação em outros projetos. Além disso, essa estruturação agiliza a compilação de grandes projetos.

O sistema make

O processo de compilação e ligação de um software composto por um grande conjunto de arquivos de código-fonte pode ser complexo e trabalhoso. Uma ferramenta que permite acelerar e facilitar esse processo é o sistema `make`.

O sistema `make` permite definir regras de dependência para a compilação de um grande conjunto de arquivos de códigos-fonte. Usando esse sistema, o processo de compilação pode ser muito agilizado, pois somente os arquivos necessários são compilados e ligados para reconstruir o arquivo executável final.

Ao ser invocado, o comando `make` procura um arquivo `Makefile` no diretório local, contendo regras para a construção dos programas. O conteúdo típico de um arquivo `Makefile` pode ser visto neste arquivo de exemplo (extraído desta página [http://vergil.chemistry.gatech.edu/resources/programming/c-tutorial/make.html]). A construção de arquivos `Makefile` está além do escopo deste curso. Mais informações podem ser facilmente obtidas na Internet [http://www.google.com/search?q=tutorial+make] e no manual do make [http://www.gnu.org/software/make/manual].

Depuração

O primeiro passo para a depuração de um programa executável é compilá-lo de forma a incluir todos os símbolos necessários ao processo de depuração (nomes das variáveis, referências às linhas do código fonte, etc):

```
~> gcc -g -o pi pi.c -lm
```

O depurador padrão para a linguagem C no Linux é o gdb - *GNU Debugger*. O gdb é um depurador em modo texto, com muitas funcionalidades mas relativamente pesado de usar. Vários exemplos de uso do gdb podem ser encontrados nos links abaixo:

- Debugging with GDB [http://www.delorie.com/gnu/docs/gdb/gdb_toc.html]
- A GDB tutorial [http://www-2.cs.cmu.edu/%7Egilpin/tutorial/]
- Guide to faster, less frustrating debugging [http://heather.cs.ucdavis.edu/%7Ematloff/UnixAndC/CLanguage/Debug.html]
- Using GNU's GDB debugger [http://www.dirac.org/linux/gdb/]

Este arquivo apresenta um exemplo simples de uso do `gdb` para analisar a execução do código executável de `pi.c`.

Várias interfaces gráficas foram construidas para facilitar o uso do `gdb`. Entre eles, o mais popular é o Data Display Debugger [http://www.gnu.org/software/ddd/], ou `ddd`. Na verdade, ele constitui uma interface para vários depuradores, como gdb, jdb (Java), Perl, Python, PHP, etc.

Outra ferramenta útil para auxiliar a depuração de programas é o utilitário `strace`, que permite listar as chamadas de sistema efetuadas por um executável qualquer. Por não precisar de opções especiais de compilação, nem do código-fonte do executável, é uma ferramenta útil para investigar o comportamento de executáveis de terceiros. Além disso, o `strace` pode analisar processos já em execução (através da

opção `-p`). Este arquivo traz um exemplo de saída do utilitário `strace` em uma execução do comando `ls`.

De forma similar, o comando `ltrace` gera uma listagem seqüencial de todas as chamadas de biblioteca geradas durante a execução de um programa.

Além da depuração propriamente dita, em busca de erros, por vezes torna-se necessário analisar o comportamento temporal do programa. Para realizar essa análise pode-se utilizar o *GNU Profiler* (`gprof`). O `gprof` permite verificar:

- o tempo dispendido em cada função
- o grafo de chamadas
  - que funções são chamadas por que funções
  - que funções chamam outras funções
- quantas vezes cada função é chamada
- ...

Para realizar o profiling de um executável, é necessário incialmente compilá-lo com o flag adequado (`-pg`) e em seguida executá-lo:

```
~> gcc -pg -g -o pi pi.c -lm
~> pi
```

A execução irá gerar um arquivo binário `gmon.out`, que contém os dados de profiling. Esse arquivo é usado pelo utilitário `gprof` para gerar as estatísticas desejadas:

```
~> gprof pi gmon.out
```

Um exemplo de relatório de saída do programa `gprof` pode ser encontrado neste arquivo. Mais detalhes e opções de relatório podem ser obtidas no manual GNU gprof [http://www.gnu.org/software/binutils/manual/gprof-2.9.1/].

Tratamento de erros

A maior parte das chamadas de sistema e funções UNIX retorna erros na forma de códigos numéricos, que são descritos nas páginas de manual das chamadas e funções (vide `man fopen` para um bom exemplo). Normalmente, uma chamada com erro retorna o valor -1 e ajusta a variável global inteira `errno` para o código do erro. Além disso, as seguintes funções podem ser úteis na interpretação dos erros:

- `perror (char * msg)` : imprime na saída de erro (`stderr`) a mensagem `msg` seguida de uma descrição do erro encontrado.
- `strerror(int errnum)` : retorna a descrição do erro indicado por `errnum`.

Além disso, alguns erros de execução, como operações matemáticas inválidas (divisão por zero, etc) ou violações de acesso à memória (ponteiros inválidos, etc) pode ser interceptados e tratados pelo programa, sem que seja necessária sua finalização. Esses erros geram sinais que são enviados ao

processo em execução, que pode interceptá-los e tratá-los de forma a contornar o erro e continuar funcionando.

Gerência de versões

O padrão para gerência de versões de software em sistemas abertos é o CVS (Concurrent Versions System). O CVS é o sistema mais usado em plataformas UNIX. Alguns links para CVS:

- CVS-Home - Introduction tyo CVS [https://www.cvshome.org/docs/blandy.html]
- Getting Started with CVS [http://www.linux.ie/articles/tutorials/cvs.php]

Ferramentas diversas

- `splint`: efetua verificações sintáticas mais estritas que o compilador C, permitindo encontrar erros frequentes de programação e vulnerabilidades de segurança, entre outros problemas.
- `strip`: permite remover os símbolos e código não usado de um executável ou arquivo-objeto, diminuindo consideravelmente seu tamanho (mas impedindo futuras depurações no mesmo).
- `diff`: permite comparar dois arquivos ou diretórios (recursivamente), apontando as diferenças entre seus conteúdos. Muito útil para comparar diferentes versões de árvores de código fonte.
- `patch` : permite aplicar um arquivo de diferenças (gerado pelo comando `diff`) sobre uma árvore de arquivos, modificando os arquivos originais de forma a obter uma nova árvore. Muito usado para divulgar novas versões de códigos-fonte muito grandes.
- `grep`: permite encontrar linhas em arquivos contendo uma determinada string ou expressão regular. Pode ser muito útil para encontrar trechos de código específicos em grandes volumes de código.
- `valgrind`: poderoso depurador de uso da memória, permite encontrar erros relacionados à alocação dinâmica de memória.

Documentação online

O sistema UNIX implementa um sistema de documentação *on-line* simples, mas bastante útil e eficiente, chamado **páginas de manual** (*man pages*). As páginas de manual estão estruturadas em sessões:

- Sessão 1: Comandos do usuário.
- Sessão 2: Chamadas ao sistema
- Sessão 3: Bibliotecas e funções standard
- Sessão 4: Descrição de dispositivos e formatos de arquivos de dados
- Sessão 5: Formato de arquivos de configuração
- Sessão 6: Jogos
- Sessão 7: Diversos
- Sessão 8: Comandos de administração do sistema

O acesso às páginas de manual é normalmente efetuado através do comando `man`. Assim, `man ls` apresenta a página de manual do comando `ls`, enquanto `man man` apresenta a página de manual do próprio comando `man`. Ambientes gráficos normalmente oferecem ferramentas mais versáteis para

consulta às páginas de manual.

Além do comando `man`, outros comandos são úteis para a busca de informações no sistema:

- `whatis`, `apropos` : para localizar páginas de manual que contenham informações sobre algum assunto específico.
- `locate` : para localizar arquivos no sistema
- `which` : para informar onde se encontra um determinado executável
- `info` : sistema de informações mais completo que o `man`, mas não disponível em todas as plataformas UNIX.

Alguns links

- YoLinux Tutorial - Software Development on Linux [http://www.yolinux.com/TUTORIALS/LinuxTutorialSoftwareDevelopment.html]
- An Introduction to GCC [http://www.network-theory.co.uk/docs/gccintro/]
- Linux Program Library HowTo [http://www.dwheeler.com/program-library/Program-Library-HOWTO/index.html]
- Linux Standard Base - Padrões de Compatibilidade em Linux [http://www.linuxbase.org/]

Atividades

1. Escreva e compile o arquivo `pi.c` (cálculo de Pi através da série de Gregory) usando compilação estática e dinâmica. Compare o tamanho final dos executáveis e os símbolos internos de cada executável (comando `nm`).
2. Construa uma biblioteca de operações aritméticas entre pares de inteiros, que implemente as operações abaixo indicadas. Devem ser gerados o arquivo fonte (`.c`), de cabeçalho (`.h`), a biblioteca estática (`.a`) e a biblioteca dinâmica (`.so`).
   - `int soma (int, int)`
   - `int subt (int, int)`
   - `int mult (int, int)`
   - `int divi (int, int)`
3. Construa um pequeno programa que use a biblioteca do exercício anterior, ligando-a (a) estaticamente e (b) dinamicamente com seu código.
4. Efetue a execução passo-a-passo do programa escrito na questão anterior, usando o `ddd` ou diretamente o `gdb`.
5. Faça o profiling do programa escrito na questão 3, usando o `gprof`.
6. Identifique todos os arquivos abertos durante a execução do comando de sistema `uptime`.

unix/desenvolvimento_em_linux.txt · Last modified: 2008/05/09 14:18 (external edit)