

Compilação

1. Linguagens compiladas e interpretadas

As pessoas geralmente descrevem as linguagens de programação como compiladas, o que significa que os programas são traduzidos para a linguagem de máquina e depois executados pelo hardware, ou interpretados, o que significa que os programas são lidos e executados por um intérprete de software. Por exemplo, **C** é considerado uma linguagem compilada e **Python** é considerada uma linguagem interpretada. Mas a distinção nem sempre é clara.

Primeiro, muitos idiomas podem ser compilados ou interpretados. Por exemplo, existem intérpretes C e compiladores Python. Segundo, existem linguagens como **Java** que usam uma abordagem híbrida, compilando programas em uma linguagem intermediária e executando o programa traduzido em um intérprete. Java usa uma linguagem intermediária chamada **Java bytecode**, que é semelhante à linguagem de máquina, mas é executada por um intérprete de software, a Java virtual machine (**JVM**).

Portanto, ser compilado ou interpretado não é uma característica intrínseca de uma linguagem; no entanto, existem algumas diferenças gerais entre linguagens compiladas e interpretadas.

2. Tipos estáticos

Muitas linguagens interpretadas suportam tipos dinâmicos, mas as linguagens compiladas geralmente são limitadas a tipos estáticos. Em uma linguagem de tipo estaticamente, você pode dizer olhando para o programa a que tipo cada variável se refere. Em uma linguagem de tipo dinâmico, você nem sempre sabe o tipo de uma variável até que o programa esteja em execução. Em geral, “**estático**” se refere a coisas que acontecem em tempo de compilação (enquanto o programa está sendo compilado) e “**dinâmico**” se refere a coisas que acontecem em tempo de execução (quando o processador está executando o código em memória) .

Por exemplo, em Python, você pode escrever uma função como esta:

```
def add (x, y):  
    return x + y
```

Olhando para este código, você não pode dizer que tipo `x` e `y` se referem no tempo de execução. Essa função pode ser chamada várias vezes, a cada chamada com valores com tipos diferentes. Quaisquer valores que suportem o operador de adição funcionarão; quaisquer outros tipos causarão uma exceção ou um **"runtime error"**.

Na linguagem C, você escreveria a mesma função assim:

```
int add (int x, int y) {  
    return x + y;  
}
```

A primeira linha da função inclui "**declarações de tipo**" para os parâmetros e o valor de retorno: `x` e `y` são declarados como inteiros, o que significa que podemos verificar em tempo de compilação se o operador de adição é legal para esse tipo. O valor de retorno também é declarado como um número inteiro.

Por causa dessas declarações, quando essa função é chamada em outra parte do programa, o compilador pode verificar se os argumentos fornecidos têm o tipo certo e se o valor de retorno é usado corretamente.

Essas verificações ocorrem antes do início da execução do programa, para que os erros possam ser encontrados mais rapidamente. Mais importante, erros podem ser encontrados em partes do programa que nunca foram executadas. Além disso, essas verificações não precisam ocorrer no tempo de execução, que é um dos motivos pelos quais as linguagens compiladas geralmente apresentam a execução mais rápida que as linguagens interpretadas.

Declarar tipos em tempo de compilação também economiza espaço. Em linguagens dinâmicas, os nomes de variáveis são armazenados na memória enquanto o programa é executado e geralmente são acessíveis pelo programa. Por exemplo, em Python, a função interna `locals` retorna um dicionário que contém nomes de variáveis e seus valores. Aqui está um exemplo em um intérprete Python:

```
>>> x = 5  
>>> print locals()  
{'x': 5, '__builtins__': <módulo '__builtin__' (built-in)>,  
'__name__': '__main__', '__doc__': None, '__package__': None}
```

Isso mostra que o nome da variável é armazenado na memória enquanto o programa está em execução (junto com outros valores que fazem parte do ambiente de tempo de execução padrão).

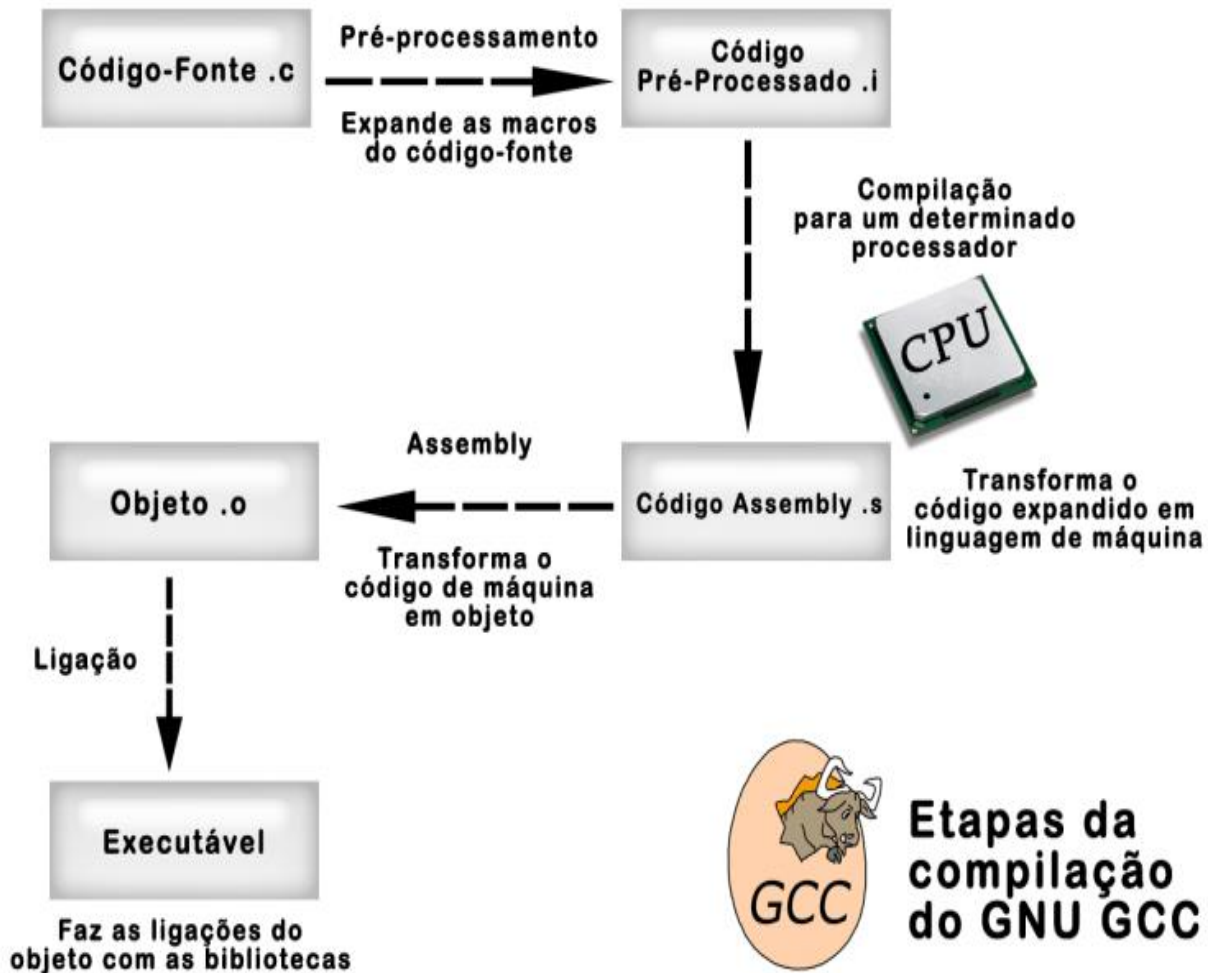
Em linguagens compiladas, os nomes de variáveis existem no tempo de compilação, mas não no tempo de execução. O compilador escolhe um local para cada variável e registra esses locais como parte do programa compilado. A localização de uma variável é chamada de "endereço". No tempo de execução, o valor de cada variável é armazenado em seu endereço, mas os nomes das variáveis não são armazenados (a menos que sejam adicionados pelo compilador para fins de depuração).

3. O processo de compilação

Como programador, você deve ter um modelo mental do que acontece durante a compilação. Se você entender o processo, ele ajudará a interpretar mensagens de erro, depurar seu código e evitar armadilhas comuns.

As etapas de compilação são:

1. **Pré-processamento:** C é uma das várias linguagens que incluem "**diretivas de pré-processamento**" que entram em vigor antes da compilação do programa. Por exemplo, a diretiva `#include` faz com que o código-fonte de outro arquivo seja inserido no local da diretiva.
2. **Análise:** durante a análise, o compilador lê o código-fonte e cria uma representação interna do programa, denominada "**árvore de sintaxe abstrata**". Os erros detectados durante esta etapa geralmente são erros de sintaxe.
3. **Verificação estática:** O compilador verifica se as variáveis e os valores têm o tipo certo, se as funções são chamadas com o número e o tipo certo de argumentos, etc. Os erros detectados durante esta etapa são às vezes chamados de erros "**semânticos estáticos**".
4. **Geração de código:** O compilador lê a representação interna do programa e gera código de máquina ou "bytecode".
5. **Vinculação (Linking):** se o programa usar valores e funções definidos em uma biblioteca, o compilador precisará encontrar a biblioteca apropriada e incluir o código necessário.
6. **Otimização:** Em vários pontos do processo, o compilador pode transformar o programa para gerar código que é executado mais rapidamente ou usa menos espaço. A maioria das otimizações são mudanças simples que eliminam desperdícios óbvios, mas alguns compiladores realizam análises e transformações sofisticadas.



O compilador mais utilizado no Linux é a Coleção de Compiladores GNU GCC. Ele compila códigos C ANSI, bem como C++, Java e Fortran. O GCC suporta vários níveis de checagem de erros nos códigos-fonte, produz informações de debug e pode ainda otimizar o arquivo objeto produzido.

Normalmente, quando você executa o **gcc**, ele executa todas essas etapas e gera um arquivo executável. Por exemplo, aqui está um programa C mínimo:

```
#include <stdio.h>
int main ()
{
    printf ("Hello world!\n");
}
```

Se você salvar esse código em um arquivo chamado `hello.c` , poderá compilar e executá-lo assim:

```
$ gcc hello.c
$ ./a.out
```

Por padrão, o gcc armazena o código executável em um arquivo chamado `a.out` (que originalmente significava "saída do assembler").

A segunda linha executa o executável. O prefixo `./` diz ao shell para procurá-lo no diretório atual.

Geralmente, é uma boa idéia usar o sinalizador `-o` para fornecer um nome melhor para o executável:

```
$ gcc hello.c -o hello
$ ./hello
```

Pré-Processamento

O pré-processamento é responsável por expandir as macros e incluir os arquivos de cabeçalho no arquivo fonte. O resultado é um arquivo que contém o código fonte expandido. O pré-processamento do `hello.c` irá gerar um arquivo em C com mais de 700 linhas de código expandido.

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

```
$ gcc -E hello.c -o hello.i
```

Veja que o arquivo preprocessado `hello.i` gerou 841 linhas de código pré-processado:

```
$ cat hello.i | wc
    731    1917   16319
```

Compilação no GCC

O próximo estágio chamado de "compilação propriamente dita" é responsável por traduzir o código-fonte pré-processado em linguagem de montagem (assembly ou código de máquina simbólico) para um processador específico.

Para que você veja o resultado do estágio de compilação, a opção `"-S"` (maiúsculo) do gcc gera o código de montagem para um processador específico.

O código abaixo é o exemplo acima já pré-processado que foi compilado em assembly para o processador Intel Xeon:

```
# gcc -S hello.i
# cat hello.s

.file "hello.c"
.text
.section .rodata
.LC0:
.string "Hello world!"
.text
.globl main
.type main, @function

main:
.LFB0:
.cfi_startproc
endbr64
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
leaq .LC0(%rip), %rdi
call puts@PLT
movl $0, %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
```

```
.LFE0:
    .size main, .-main
    .ident      "GCC: (Ubuntu 9.3.0-10ubuntu2) 9.3.0"
    .section    .note.GNU-stack,"",@progbits
    .section    .note.gnu.property,"a"
    .align 8
    .long 1f - 0f
    .long 4f - 1f
    .long 5

0:
    .string     "GNU"

1:
    .align 8
    .long 0xc0000002
    .long 3f - 2f

2:
    .long 0x3

3:
    .align 8

4:
```

Assembler do Código no GCC

O estágio seguinte é chamado de assembler. Nesta etapa o gcc converte o código de montagem de um processador específico em um arquivo objeto. Se neste código existirem chamadas externas de funções, o gcc deixa seus endereços indefinidos para serem preenchidos posteriormente pelo estágio de ligação.

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

```
# as hello.s -o hello.o
```

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

O resultado será um arquivo “`hello.o`” que contém instruções de máquina do exemplo com a referência indefinida para a função externa `printf`.

A linha “`call printf`” do código de montagem informa que esta função está definida em uma biblioteca e deverá ser chamada durante o processamento.

Linker do código com as bibliotecas

O último estágio chamado de ligação, ou linker, é responsável por ligar os arquivos objeto para criar um arquivo executável. Ele faz isto 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 do C para serem executados.

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

No primeiro caso, o programa utilizará bibliotecas de forma compartilhada e ficará dependente delas para funcionar. Este esquema economiza recursos, pois uma biblioteca utilizada por muitos programas precisa ser carregada somente uma vez na memória. O tamanho do executável será pequeno.

No segundo caso, o programa é independente, uma vez que as funções necessárias estão no seu código. Este esquema permite que quando houver uma mudança de versão de biblioteca o programa não será afetado. A desvantagem será o tamanho do executável e necessidades de mais recursos.

Internamente a etapa de ligação é bem complexa, mas o GCC faz isto de forma transparente através do comando:

```
$ gcc hello.o -o hello
```

O resultado será um executável chamado *hello*.

```
# ./hello  
Hello world!
```

Este programa foi ligado às bibliotecas de forma compartilhada. O comando `ldd` informa quais são as bibliotecas ligadas ao executável:

```
$ ldd hello  
linux-vdso.so.1 (0x00007ffcdf3c7000)  
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f50eb213000)  
/lib64/ld-linux-x86-64.so.2 (0x00007f50eb41c000)
```


As bibliotecas *libc.so.6* e *ld-linux.so.2* são referenciadas dinamicamente pelo programa *hello*. A opção “-static” do compilador copia as funções externas das bibliotecas para o executável:

```
$ gcc -static hello.c -o hello1
```

O comando *ldd* informará que o executável *hello1* não tem ligações dinâmicas:

```
$ ldd hello1
not a dynamic executable
```

A diferença no tamanho dos executáveis é muito grande. O programa *hello* que utiliza ligações dinâmicas tem 16.688 bytes. O *hello1* tem 871.696 bytes. Portanto, somente utilize a cópia das bibliotecas para o executável se for estritamente necessário.

4. Compreendendo erros

Agora que conhecemos as etapas do processo de compilação, é mais fácil entender as mensagens de erro. Por exemplo, se houver um erro em uma diretiva *#include*, você receberá uma mensagem do pré-processador:

```
hello.c:1:10: fatal error: stdiooo.h: no such file or directory
  1 | #include <stdiooo.h>
    |           ^~~~~~
Compilation terminated.
```

Se houver um erro de sintaxe, você receberá uma mensagem do compilador:

```
hello.c: in function 'main':
hello.c:4:29: error: expected ';' before '}' token
  4 |         printf("Hello world!\n")
    |                                     ^
    |                                     ;
  5 |     }
    |     ~
```

Se você usar uma função que não está definida em nenhuma das bibliotecas padrão, receberá uma mensagem do vinculador:

```
hello.c: In function 'main':
hello.c:4:5: warning: implicit declaration of function 'printf';
did you mean 'printf'? [-Wimplicit-function-declaration]
    4 |     printf("Hello world!\n");
      |     ^~~~~~
      |     printf
/usr/bin/ld: /tmp/cc8pFlcP.o: in function `main':
hello.c:(.text+0x15): undefined reference to `printf'
collect2: error: ld returned 1 exit status
```

`ld` é o nome do linker do GCC.

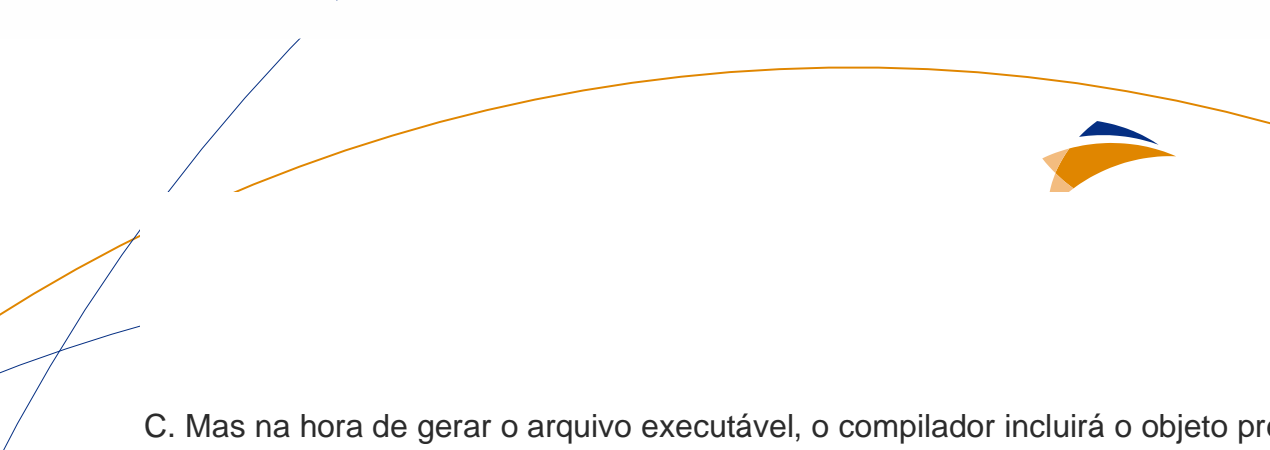
Depois que o programa é iniciado, C faz muito pouca verificação de tempo de execução; portanto, você provavelmente verá apenas alguns erros de tempo de execução. Se você dividir por zero ou executar outra operação ilegal de ponto flutuante, receberá uma "exceção de ponto flutuante". E se você tentar ler ou gravar um local incorreto na memória, receberá uma "segmentation fault" (ou falha de segmentação).

BIBLIOTECA C

Uma biblioteca da linguagem C é composta por dois tipos de arquivos:

- **Cabeçalho** (*header*) – tem extensão `.h` e possui a definição de funções, macros, variáveis e/ou constantes. Este tipo de arquivo usa a linguagem C e é essencial para a compilação dos programas que usam a biblioteca.
- **Objeto pré-compilado** (binário) – pode ter extensão `.a` (biblioteca estática) ou `.so` (biblioteca compartilhada) e corresponde ao executável da biblioteca (os cabeçalhos apenas declaram as funções).

Por exemplo, suponha um cabeçalho com as definições das funções de E/S. Quando um programa precisa acessar um arquivo, este cabeçalho é incluído no código. O cabeçalho será importante para o compilador encontrar todas as referências feitas no programa



C. Mas na hora de gerar o arquivo executável, o compilador incluirá o objeto pré-compilado correspondente.

GNU C Library (glibc)

A GNU C Library, ou **glibc**, é a biblioteca padrão C do **projeto GNU**. A glibc adota vários padrões como C11 para a linguagem C e POSIX.1-2008 (Portable Operating System Interface) para os cabeçalhos do sistema operacional. Portanto, a glibc possui a implementação de diferentes propostas para a bibliotecas C.

No Linux, os cabeçalhos da biblioteca C costumam ser instalados em `/usr/include`, enquanto os cabeçalhos do kernel são colocados no diretório `/usr/src/<versão do Linux>/` como, por exemplo, `/usr/src/linux-headers-4.15.0-45-generic`.

Tipos de biblioteca

No Linux, uma biblioteca pode ser:

- **Estática** – o código da biblioteca é copiado e inserido no executável (binário) do programa. Isto significa que, mesmo que a biblioteca seja alterada e/ou descontinuada, isto não afetará o programa. Este tipo de biblioteca tem extensão `“.a”`.
- **Dinâmica (compartilhada)** – o código da biblioteca não é inserido no executável do programa. O binário tem apenas referências à biblioteca desejada (assim o arquivo executável tem um tamanho menor). Estas referências são resolvidas durante a execução do programa. Além disso, a biblioteca pode ser usada (compartilhada) por vários programas. Este tipo de biblioteca tem extensão `“.so”`.

Os binários das bibliotecas compartilhadas do sistema ficam em `/lib`, enquanto os binários das bibliotecas dos programas dos usuários (não essenciais ao sistema) ficam em `/usr/lib` ou `/usr/local/lib`.

Criando uma biblioteca

A melhor forma de entender como funcionam as bibliotecas no Linux é através de um exemplo. Em primeiro lugar, vamos definir um cabeçalho (*header*) que terá a declaração das funções. Vamos chamar este cabeçalho de `nova_bib.h`.

```
#ifndef _NOVA_BIBLIOTECA
#define _NOVA_BIBLIOTECA

#include <stdio.h>
int fatorial(int);
int fibonacci(int);

#endif
```

Inicialmente, o cabeçalho verifica se a macro `_NOVA_BIBLIOTECA` ainda não foi definida no código. Se não foi, então a macro é definida e o resto do código é lido. Se já foi definida, o resto do código é ignorado. Isto evita múltiplas definições da mesma biblioteca (cada cabeçalho precisa ter um nome diferente no início do código). Na terceira linha, é incluído o cabeçalho `stdio.h` que contém funções, macros e tipos de dados necessários para as operações de E/S como, por exemplo, a função `printf()`. Em seguida são declaradas duas funções, `fatorial()` e `fibonacci()`, que recebem um parâmetro inteiro e retornam um valor inteiro. A última linha define o término da macro `_NOVA_BIBLIOTECA`.

Em segundo lugar, vamos criar o arquivo `nova_bib.c` com as funções `fatorial()` e `fibonacci()`. Note que o novo cabeçalho é incluído na primeira linha. Em seguida, temos os códigos das duas funções.

```
#include "nova_bib.h"

int fatorial(int num)
{
    int i;
    int fat = 1;

    for (i = 2; i <= num; i++)
        fat = fat * i;

    return fat;
}

int fibonacci(int num)
{
    int i;
    int fib = 0;
    int fib_a = 0;
    int fib_b = 1;

    for (i = 2; i <= num; i++)
    {
        fib = fib_a + fib_b;
        fib_a = fib_b;
        fib_b = fib;
    }

    return fib;
}
```

Em terceiro lugar, vamos definir o arquivo `teste_lib.c` que chama as funções da biblioteca (poderia ser apenas uma das funções). Inicialmente é incluído o cabeçalho `nova_bib.h` que possui as funções `fatorial()` e `fibonacci()`. Note que o cabeçalho está entre aspas. Isto significa que o sistema vai primeiro procurar o arquivo no diretório corrente e depois, se não encontrar o cabeçalho, vai percorrer a lista dos diretórios fornecida. Quando se usa “<” e “>” no lugar das aspas, o cabeçalho é procurado primeiro na lista. Note também que o cabeçalho `stdlib.h` que possui a função `atoi()` (converte uma string em um inteiro) não é especificada. Isto ocorre porque o compilador a inclui automaticamente.

Quando o usuário executar o programa `teste_lib.c`, ele deverá informar um número. Se o número é fornecido na linha de comando, as funções `fatorial()` e `fibonacci()` são chamadas e o número recebido é passado como parâmetro. Note que o programa assume que foi passado um parâmetro numérico, quando o correto é verificar o tipo de parâmetro passado antes de chamar as funções.

```
#include "nova_bib.h"
```

```
int main(int argc, char *argv[])
{
    int num;

    if (argc != 2)
    {
        printf("Forneca um numero\n");
        return 1;
    }
    else
        num = atoi(argv[1]);

    printf("\n*** fatorial\n");
    int fat = fatorial(num);

    printf("Fatorial de %d = %d\n", num, fat);
    printf("\n*** fibonacci\n");

    int fib = fibonacci(num);
    printf("Soma dos %d primeiros numeros de Fibonacci = %d\n\n",
num, fib);
}
```

A seguir, vamos ver como gerar o executável com uma biblioteca estática:

1. Compilar a nova biblioteca sem linkar (apenas cria o arquivo objeto).

```
$ gcc -c nova_bib.c -o nova_bib.o
```

2. Criar o repositório da biblioteca com o aplicativo **ar**. Para cada arquivo adicionado ao repositório, o **ar** inclui informações sobre permissões, data e identificação do dono e do grupo. No nosso exemplo, o repositório criado é chamado de `libnova_bib.a` e só tem um arquivo (se existissem outros arquivos, os nomes seriam colocados após `nova_bib.o`).

```
$ ar rc libnova_bib.a nova_bib.o
```

onde a opção **r** adiciona/substitui arquivos no repositório e a opção **c** cria o repositório (se já existe um arquivo com o mesmo nome, ele é deletado).

3. Para executar o arquivo `teste_lib.c` é preciso gerar o executável com a nova biblioteca.

```
$ gcc teste_lib.c -o teste_lib -L. -lnova_bib
```

onde a opção **L.** inclui o diretório corrente na lista de busca pela biblioteca e a opção **-lnova_bib** – diz para linkar a biblioteca `libnova_bib.a` ao programa. Note que nem o prefixo “lib” e nem a extensão “.a” são informados no comando, pois fazem parte do padrão do nome da biblioteca. Caso o nome da biblioteca não seguisse o padrão, o comando usado seria

```
$ gcc teste_lib.c -o teste_lib -L. biblioteca
```

4. Para ver as bibliotecas compartilhadas que são usadas pelo arquivo `teste_lib`, basta digitar:

```
$ ldd teste_lib
```

A saída abaixo mostra que estão sendo usadas três bibliotecas na execução do arquivo `teste_lib` (além da biblioteca estática `libnova_bib.a`)

```
linux-vdso.so.1 => (0x00007ffffade4000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007fe1611d3000)
/lib64/ld-linux-x86-64.so.2 (0x00007fe1615bc000)
```

5. Caso você queira distribuir a nova biblioteca, basta fornecer os arquivos `nova_bib.h` e `libnova_bib.a`.

CHAMADAS DE SISTEMA

Descrição

As chamadas de sistemas são funções (interfaces) usadas pelos aplicativos para solicitar a execução de algum serviço ao **kernel** do sistema operacional. Por isso, as chamadas de sistemas são instruções com maior privilégio quando comparadas às outras instruções. Com as chamadas de sistemas é possível, por exemplo, definir acesso a recursos de baixo nível como alocação de memória, periféricos e arquivos. Além disso, são as chamadas de sistemas que permitem a criação e a finalização de **processos**. Quando a execução de uma chamada de sistema é solicitada, o sistema operacional salva todo o contexto do processo (para continuar mais tarde de onde parou), verifica as permissões envolvidas no pedido e autoriza (se for o caso) o processador a executar o serviço solicitado.

Quando o processador termina a execução da chamada de sistema, o sistema operacional retorna o controle para o processo, colocando-o novamente na fila de processos prontos para a execução.

No **Linux**, *kernel* 4.11, existem cerca de 400 chamadas de sistemas disponibilizadas na **biblioteca C** (a biblioteca faz a interface entre o aplicativo e o kernel).

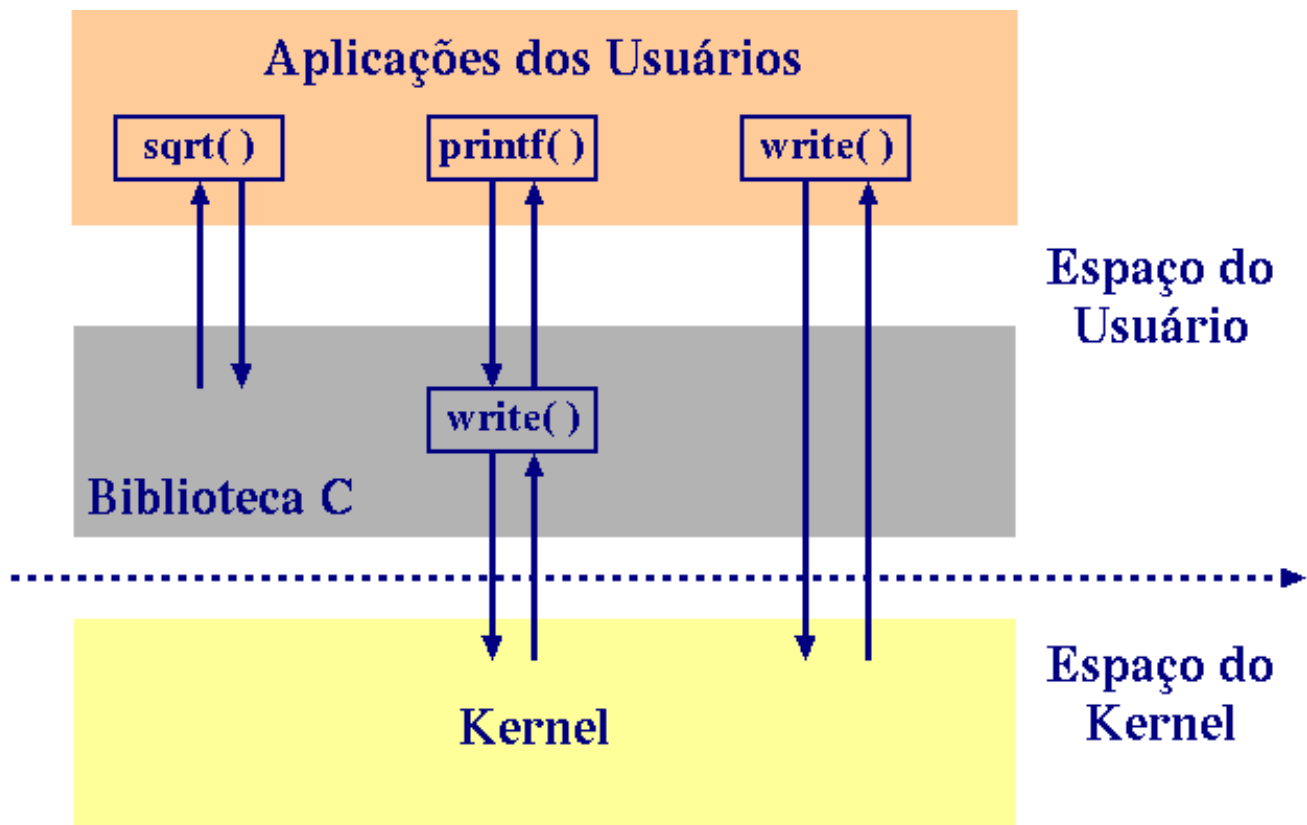
Programação

O programador normalmente não utiliza as chamadas de sistema no seu código. Ele utiliza uma função de biblioteca que é transformada em uma ou mais chamadas de sistema quando o código executável é gerado e há necessidade de pedir um serviço ao kernel. Por exemplo, a função `printf()` da Linguagem C é mapeada na chamada `write()` para escrever em um arquivo. Já a função matemática `sqrt()` não precisa de privilégios especiais para ser executada e não há necessidade de pedir ajuda ao kernel.

O programador pode usar as chamadas de sistema no seu código se, por exemplo, está usando a Linguagem C. Com isto, ele ganha rapidez na compilação do programa (não precisa fazer a conversão), mas diminui a portabilidade do código (o formato das chamadas pode variar com as arquiteturas).

Lista

As chamadas de sistema do **Linux** podem ser encontradas no arquivo `/usr/include/unistd.h` ou podem ser obtidas com o comando `man syscalls`



O exemplo a seguir imprime a data e o horário a cada 5 segundos.

```
1.  #include<unistd.h>
2.  #include<stdio.h>
3.  #include<time.h>
4.  #include<signal.h>
5.  void mensagem(int signum)
6.  {
7.      time_t tp;
8.      time(&tp);
9.      printf("%s", ctime(&tp));
10. }
11.
12. void main( )
13. {
14.     signal(SIGALRM, mensagem);
15.     printf("*** inicio do programa\n");
16.     while (1)
17.     {
18.         alarm(5);
19.         pause( );
20.     }
21. }
```


- **Linhas de 1 a 4** : é feita a inclusão dos arquivos *headers* da **biblioteca C**.
 - a) **unistd.h** – cabeçalho com as chamadas de sistema.
 - b) **stdio.h** – cabeçalho com a definição das funções de E/S dos arquivos como, por exemplo, *printf()*.
 - c) **time.h** – cabeçalho com as funções para manipular datas e horas.
 - d) **signal.h** – cabeçalho com a definição dos **sinais**.
- **Linhas de 5 a 10** : a função *mensagem()* é chamada quando um sinal SIGALRM é capturado pelo processo. Esta função apenas imprime o dia e a hora local obtidos com a chamada de sistema *time()* e formatada com a função *ctime()*.
- **Linha 11** : início da função *main()* cujos comandos vão da linha 13 a linha 19.
- **Linha 13** : usa a chamada *signal()* para definir que a função *mensagem()* deve ser executada quando o **sinal** SIGALRM é capturado pelo processo.
- **Linhas de 15 a 19** : define um loop infinito.
- **Linha 17** : usa a chamada *alarm()* para definir que um **sinal** SIGALRM deve ser entregue ao processo em 5 segundos.
- **Linha 18** : usa a chamada *pause()* para suspender o processo até o recebimento de um sinal (neste exemplo, a espera é pelo sinal SIGALRM).

Abaixo, um exemplo de saída do programa.

```
*** inicio do programa
Sun Feb 3 18:08:01 2019
Sun Feb 3 18:08:06 2019
Sun Feb 3 18:08:11 2019
Sun Feb 3 18:08:16 2019
```