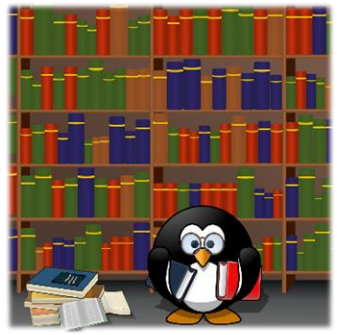
Sistemas Operacionais II

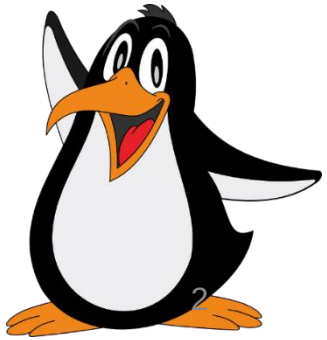
Técnicas Básicas para programação em GNU/LINUX – Parte 2

Fabricio Breve
fabricio.breve@unesp.br
https://www.fabriciobreve.com

Agenda

- Programação Defensiva
  - A Macro *assert*
  - Falhas em Chamadas de Sistema
  - Códigos de Erros em Chamadas de Sistema
  - Erros e Alocação de Recursos
- Escrevendo e Usando Bibliotecas
  - Arquivos (Bibliotecas Estáticas)
  - Bibliotecas Compartilhadas
  - Bibliotecas Padrões
  - Dependências entre Bibliotecas
  - Vantagens e Desvantagens
  - Carga e Descarga Dinâmica





Programação Defensiva

- Nesta seção veremos algumas técnicas para:
  - Encontrar *bugs* o quanto antes
  - Detectar e se recuperar de problemas durante a execução de um programa

https://devhumor.com/media/compiles-and-works-the-first-time

A macro `assert`

- Macro que recebe como parâmetro uma expressão Booleana.
  - Se a expressão for falsa o programa termina, imprimindo arquivo-fonte, linha de código e texto do local da falha.
  - Muito útil para checar consistências internamente no programa
    - Exemplos: argumentos de funções, pré-condições e pós-condições para chamadas de funções (ou métodos em C++), teste para valores de retorno não esperados.

A macro `assert`

- Também serve para “documentar” o programa.
  - Alguém lendo seu código saberá que aquela condição precisa ser sempre verdadeira, do contrário há um *bug* no programa.
- Prejudica o desempenho do programa.
  - Use a flag –DNDEBUG para remover as macros **`assert`** no pré-processamento.
  - Nunca chame funções, atribua valores à variáveis ou use operadores de modificação (ex.: ++) em expressões **`assert`**

A macro `assert`

- Nunca faça isso:

```
for (i = 0; i < 100; ++i)
        assert (faca_algumacoisa() == 0);
```

  - Se compilar com `-DNDEBUG`, a função nunca será executada.

- Faça isso:

```
for (i = 0; i < 100; ++i) {
        int status = faca_algumacoisa();
        assert (status == 0);
}
```

A macro `assert`

- Não use para validar entrada de usuário.
  - Usuários não gostam quando aplicativos terminam com mensagens de erro incompreensíveis.
  - Use apenas para checagens internas.

A macro `assert`

- Exemplos de uso:
  - Checar ponteiros nulos
    - `{assert (pointer != NULL)}`
    - Erro gerado:
      - `Assertion ‘pointer != ((void *)0)’ failed.`
    - Erro gerado ao desreferenciar um ponteiro nulo:
      - `Segmentation fault (core dumped)`
  - Checar condições em parâmetros de funções
    - Ex.: função que deve ser chamada apenas com números maiores que zero.
      - Colocar no início da função:
        - » `assert (foo > 0);`

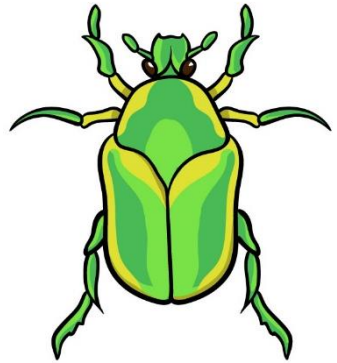
Falhas em Chamadas de Sistema

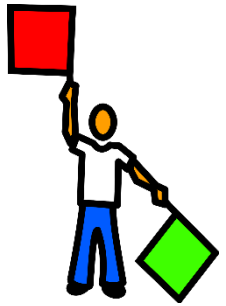
- Chamadas de Sistemas podem falhar por vários motivos:
  - Falta de recursos no sistema (ou de recursos fornecidos pelo sistema para um único programa).
    - Exemplos: alocar muita memória, escrever muito em disco, abrir muitos arquivos ao mesmo tempo.
  - Falta de permissão.
    - Exemplos: tentar escrever em arquivo somente para leitura, acessar memória de outro processo, matar programa de outro usuário.

Continua ►

Falhas em Chamadas de Sistema

- Argumentos da chamada de sistema inválidos (usuário forneceu parâmetros errados ou *bugs* do programa).
  - Exemplos: endereço de memória inválido, descritor de arquivo inexistente, tentar abrir diretório como se fosse um arquivo, passar nome de arquivo quando é esperado um diretório, etc.
- Erros provocados por hardware.
  - Exemplos: falha em dispositivo, disco não inserido, dispositivo não suporta determinada operação, etc.
- Chamada de sistema interrompida por evento externo
  - Exemplo: sinal. Neste caso cabe ao programa que faz a chamada de sistema, chamá-la novamente.





Falhas em Chamadas de Sistema

- Em programas que fazem muitas chamadas de sistema, é comum ter mais código para tratar erros do que para realizar as tarefas do fluxo normal de execução.

```
try {
  // Bloco de código para tentar
  throw exception; // Lance uma exceção quando surgir um problema
}
catch () {
  // Bloco de código para lidar com erros
}
```

Exceções...
Temos que pegar!
(Gotta Catch 'Em All!)

Códigos de Erros em Chamadas de Sistema

- A maioria das chamadas de sistema retorna zero se tudo correu bem e não-zero se um erro ocorreu.
  - Há exceções. Por exemplo: **`malloc`** retorna um ponteiro nulo para indicar falha. Sempre consulte o manual para ter certeza.
- A maioria das chamadas de sistema tem uma variável especial chamada **`errno`** que armazena informações adicionais em caso de falha.
  - Quando a chamada falha, o sistema ajusta **`errno`** para um valor que indica o que deu errado.
  - Copie o valor para outra variável, pois ele será sobrescrito na próxima chamada de sistema.

Códigos de Erros em Chamadas de Sistema

- Os valores de erro são inteiros.
  - Use macros para se referir a eles por nomes como **`EACCES`** e **`EINVAL`**
  - Inclua **`<errno.h>`** no cabeçalho.
- **`strerror`**
  - Retorna uma *string* de caracteres com a descrição do erro, útil para fornecer mensagens de erro (em inglês).
  - Inclua **`<string.h>`** para utilizá-la.

Códigos de Erros em Chamadas de Sistema

- **`perror`**
  - Imprime a descrição do erro diretamente para o fluxo **`stderr`**
  - Inclua **`<stdio.h>`** para usá-la.
- Exemplo: erro em abertura de arquivo.

```
fd = open ("arquivoentrada.txt", O_RDONLY);
if (fd == -1) {
        /* Abertura falhou. Imprima mensagem de erro e saia. */
        fprintf (stderr, "erro abrindo arquivo: %s\n", strerror (errno));
        exit (1);
}
```

Implementando o exemplo do slide anterior

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
int main()
{
        int fd;
        fd = open ("arquivoentrada.txt", O_RDONLY);
        if (fd == -1) {
                /* Abertura falhou. Imprima mensagem de erro e saia. */
                fprintf (stderr, "erro abrindo arquivo: %s\n", strerror (errno));
                exit (1);
        }
}
```

Veja em execução: https://youtu.be/_EHljh_6TvE

Erros e Alocação de Recursos

- Considere um programa que lê de um arquivo para um buffer:
  1. Alocar o buffer
  2. Abrir o arquivo
  3. Ler do arquivo para o buffer
  4. Fechar o arquivo
  5. Retornar o buffer
- Se o passo 2 falhar, é preciso desalocar o buffer alocado no passo 1 antes de retornar.
- Se o passo 3 falhar, é preciso também fechar o arquivo, antes de retornar.

```c
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

char* read_from_file (const char* filename, size_t length)
{
  char* buffer;
  int fd;
  size_t bytes_read;

  /* Alocar o buffer.  */
  buffer = (char*) malloc (length);
  if (buffer == NULL)
    return NULL;
  /* Abrir o arquivo.  */
  fd = open (filename, O_RDONLY);
  if (fd == -1) {
    /* falhou ao abrir. Desalocar o buffer antes de retornar.  */
    free (buffer);
    return NULL;
  }
  /* Ler os dados.  */
  bytes_read = read (fd, buffer, length);
  if (bytes_read != length) {
    /* leitura falhou. Desalocar o buffer e fechar fd antes de retornar.  */
    free (buffer);
    close (fd);
    return NULL;
  }
  /* Tudo bem. Feche o arquivo e retorne o buffer.  */
  close (fd);
  return buffer;
}
```

readfile.c

Escrevendo e Usando Bibliotecas

- Praticamente todo programa é ligado (*linked*) com uma ou mais bibliotecas.
  - Entrada/saída
  - Interface gráfica
  - Acesso a bases de dados
  - Etc...
- Em cada caso você deve decidir ligar (*link*) a biblioteca estaticamente ou dinamicamente.

Escrevendo e Usando Bibliotecas

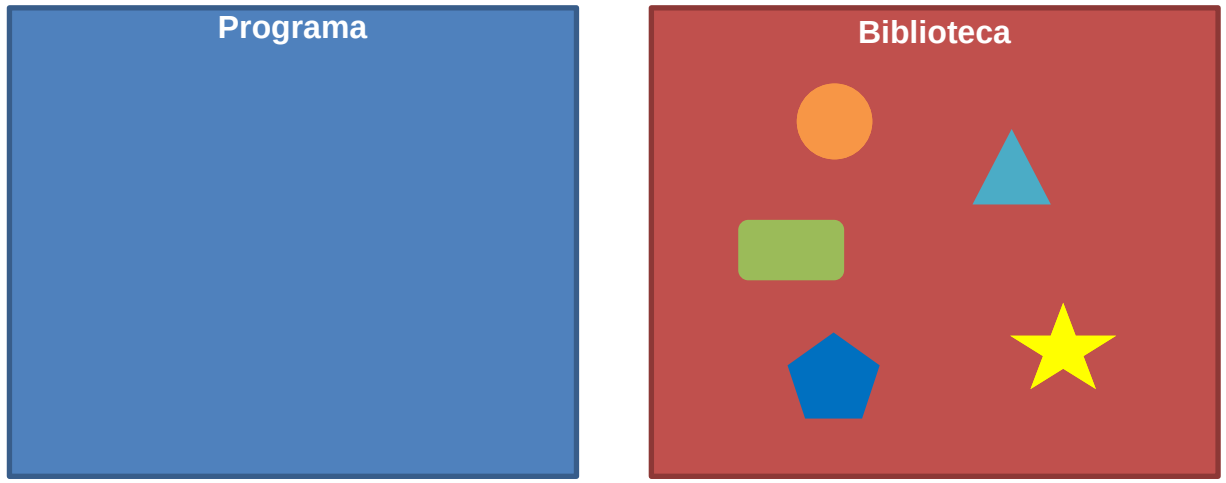
- Ligando estaticamente:
  - Programas maiores
  - Difíceis de atualizar
  - Mais fáceis de distribuir (*deploy*)
    - Tornar utilizável em outros sistemas
- Ligando dinamicamente:
  - Programas menores
  - Mais fáceis de atualizar
  - Mais difíceis de distribuir

Arquivos (Bibliotecas Estáticas)

- Um arquivo (do inglês *archive*, ou biblioteca estática) é uma coleção de arquivos objeto armazenados em um único arquivo (*file*).
  - O *gcc* procura no arquivo pelos objetos que precisa, extraindo-os e ligando-os ao programa.
  - Para criar um arquivo utilize o comando **`ar`**
    - Exemplo:
      - `ar cr libtest.a test1.o test2.o`
        - » Flag `cr` indica que `libtest.a` deve ser criado
        - » Arquivos normalmente tem extensão `.a`
        - » Para ligar esta biblioteca em algum programa utilize:
          - `gcc -o myprog myprog.o -L. -ltest`

Arquivos (Bibliotecas Estáticas)

- Quando o vinculador (*linker*) encontra um arquivo na linha de comando, procura nele todas as definições de símbolos (funções ou variáveis) que foram referenciadas nos arquivos objetos já processados que ainda não foram definidas.



Arquivos (Bibliotecas Estáticas)

- Exemplo:

```c
int f ()
{
  return 3;
}
```

test.c

```c
extern int f ();

int main ()
{
  return f ();
}
```

app.c

Exercício 1:

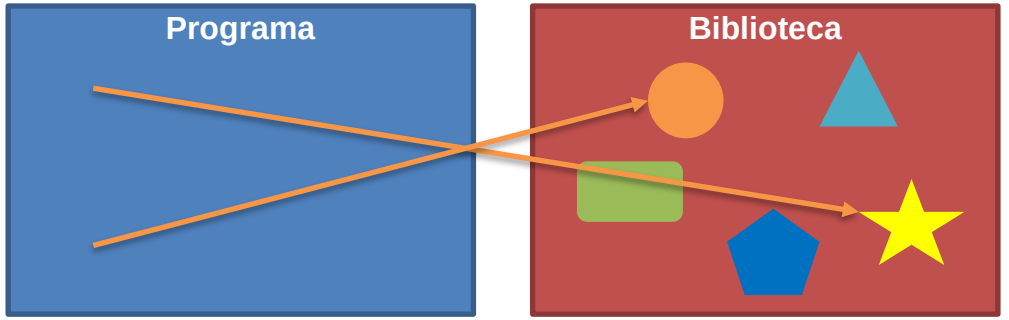
a) Compile `test.c` e coloque-o em uma biblioteca estática chamada **`libtest.a`**
b) Gere o arquivo objeto para `app.c`
c) Agora experimente liga-los com os seguintes comandos:

```
gcc -o app -L. -ltest app.o
gcc -o app app.o -L. -ltest
```

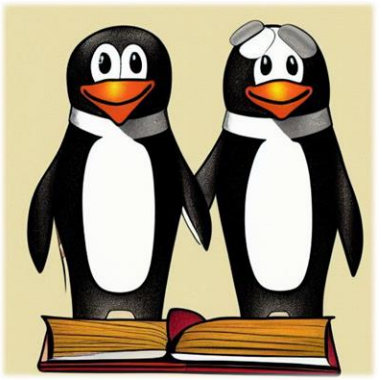
Qual funciona? Por que?

Atenção: Coloque as respostas dos exercícios de 1 a 3 no Google Classroom.

Bibliotecas Compartilhadas



- Também chamada:
  - Objeto compartilhado
  - Biblioteca ligada dinamicamente
- Agrupam arquivos objetos, tal qual bibliotecas estáticas.
- Quando é ligada em um programa, o executável **não** contém o código presente na biblioteca compartilhada.
  - Contém apenas uma referência para a biblioteca compartilhada.
- Se vários programas se ligarem à mesma biblioteca, todos referenciarão a mesma biblioteca.



Bibliotecas Compartilhadas

- Os arquivos objetos em uma biblioteca compartilhada são combinados em um único arquivo objeto.
  - Quando um programa se liga a uma biblioteca compartilhada, ele "incluirá" todo o código da biblioteca e não apenas as partes necessárias.
- Para criar uma biblioteca compartilhada, os objetos que irão formá-la devem ser compilados com a opção `-fPIC`:
  - Exemplo:
    - `gcc -c -fPIC test1.c`

Bibliotecas Compartilhadas

- Para combinar os arquivos objetos em uma biblioteca compartilhada use:
  - `gcc -shared -fPIC -o libtest.so test1.o test2.o`
    - A opção `-shared` diz ao `gcc` para produzir uma biblioteca compartilhada em vez de um executável.
    - Bibliotecas compartilhadas usam a extensão `.so` (*shared object*)
- Ligar uma biblioteca compartilhada é como ligar uma biblioteca estática.
  - Exemplo: para ligar libtest.so
    - `gcc -o app app.o -L. -ltest`

Bibliotecas Compartilhadas

- E se você tiver `libtest.a` e `libtest.so`?
  - Se não estiverem no mesmo diretório.
    - O vinculador escolhe a primeira que encontrar, procurando primeiro nos diretórios especificados em `-L`, e depois nos diretórios padrões.
  - Se estiverem no mesmo diretório.
    - O vinculador escolhe a biblioteca compartilhada.
    - Quer que ele escolha a biblioteca estática? Então indique explicitamente o caminho da biblioteca:
      - `gcc -o app app.o ./libtest.a`
      - Obs: não faça isso com bibliotecas dinâmicas!

A opção `-static` do gcc

- A opção `-static` pode ser usada para indicar ao **gcc** que ele deve ligar o executável estaticamente.
  - Sem nenhuma dependência de biblioteca em tempo de execução.
  - Sintaxe:
    - `gcc -static -o app app.o -L. -ltest`
  - Essa opção também deixa de ligar dinamicamente bibliotecas padrões do Linux e do C.

Veja um exemplo: https://youtu.be/e9E9En2fJ0o

Obs.: No Fedora pode ser necessário instalar o pacote **`glibc-static`** para poder usar `-static`

Especificando outros locais para o executável procurar por bibliotecas

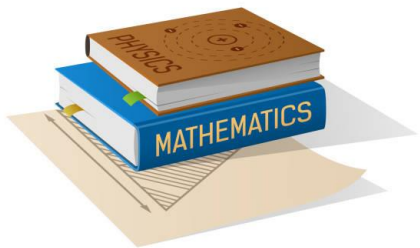
- Ao ligar um programa com uma biblioteca compartilhada, o vinculador **não** coloca o caminho completo da biblioteca no executável.
  - Coloca apenas o nome da biblioteca.
  - Ao executar o programa, o sistema procura a biblioteca em `/lib` e `/usr/lib`
  - Uma solução para buscar bibliotecas em outros locais é compilar com `-Wl,-rpath`
    - Exemplo:
      - `gcc -o app app.o -L. -ltest -Wl,-rpath,/usr/local/lib`
        - » Ao executar **app**, o sistema procurará bibliotecas compartilhadas requeridas em `/usr/local/lib`

Usando LD_LIBRARY_PATH

- Outra solução:
  - Configurar variável de ambiente `LD_LIBRARY_PATH` ao executar o programa.
    - `LD_LIBRARY_PATH` é uma lista de diretórios separada por dois pontos (`:`)
      - Exemplo:
        - » `/usr/local/lib:/opt/lib`
      - Caminho em `LD_LIBRARY_PATH` é pesquisado antes dos diretórios padrões.
    - Essa variável também indica ao vinculador para procurar nesses diretórios, além dos indicados com `-L`

Bibliotecas Padrões

- Mesmo que você não especifique nenhuma biblioteca ao ligar seu programa, é quase certo que ele usará uma biblioteca compartilhada.
  - O GCC automaticamente liga a biblioteca C padrão chamada **`libc`**
- As funções matemáticas da biblioteca C ficam em **`libm`** e não são ligadas automaticamente.
  - Exemplo: para compilar um programa que use **`sin`** e **`cos`**:
    - `gcc -o compute compute.c -lm`



Dependências de Bibliotecas

- Uma biblioteca frequentemente depende de outra
  - Exemplo:
    - A biblioteca **`libtiff`**, que contém funções para ler e gravar imagens no formato TIFF, depende de várias outras bibliotecas:
      - **`liblzma`**
      - **`libjbig`**
      - **`libjpeg`**
      - **`libz`**
      - **`libm`**

Dependências de Bibliotecas

- Exemplo: `tifftest.c`

```c
#include <stdio.h>
#include <tiffio.h>

int main (int argc, char** argv)
{
  TIFF* tiff;
  tiff = TIFFOpen (argv[1], "r");
  TIFFClose (tiff);
  return 0;
}
```

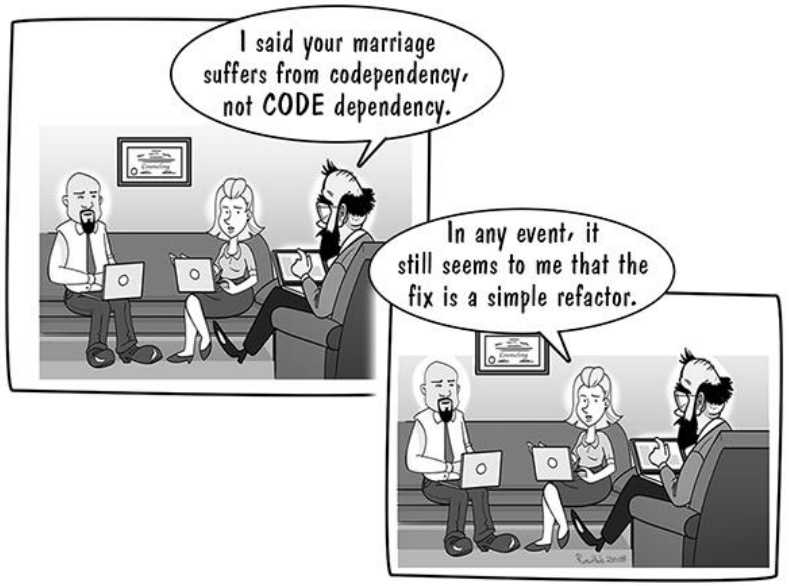
```
gcc -o tifftest tifftest.c -ltiff
ldd tifftest
```

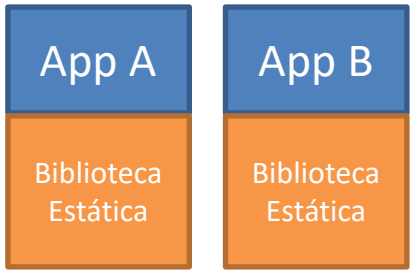
Veja em execução:
https://youtu.be/EkigN3yTJ-c

Obs. 1: Note que o LDD não acha bibliotecas fora do path
Obs. 2: No Ubuntu, pode ser necessário instalar o pacote **`libtiff-dev`**
Obs. 3: No Fedora, pode ser necessário instalar o pacote **`libtiff-devel`**

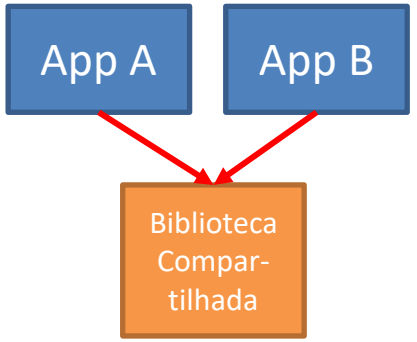
Dependências de Bibliotecas

- Ocasionalmente, duas bibliotecas podem ser mutuamente dependentes.
  - Resultado de falha de projeto.
  - Para contornar o problema, é possível oferecer um nome de biblioteca múltiplas vezes na linha de comando.
    - Exemplo:
      - `gcc -o app app.o -lfoo -lbar -lfoo`





Vantagens e Desvantagens



Biblioteca Estática

- Gasta mais espaço no sistema onde o programa está instalado.
  - Cada programa terá sua própria cópia da biblioteca ligada a ele.
- Atualizações em bibliotecas requerem atualizações nos programas.
  - Pode ser vantajoso para evitar que alguma atualização de biblioteca cause um *bug* no software.
- Não requer que o usuário instale bibliotecas.
  - Evita que o usuário tenha que instalar bibliotecas ou modificar variáveis de ambiente.

Biblioteca Compartilhada

- Economiza espaço no sistema onde o programa está instalado.
  - Vantagem aumenta quando vários programas usam mesma biblioteca.
- Usuários podem atualizar bibliotecas sem atualizar os programas que dependem delas.
  - Se vários programas usam a mesma biblioteca, basta atualizá-la e todos os programas estarão atualizados, sem necessidade de religar.
- Requer que o usuário instale bibliotecas.
  - Para instalar em `/lib` ou `/usr/lib` é necessário privilégios de *root*
  - O usuário tem um passo extra ao ter que modificar variáveis de ambiente.

Carga e Descarga Dinâmica

- É possível carregar um código dinamicamente sem liga-lo explicitamente.
  - Útil para aplicações que aceitam módulos *plug-in*, como navegadores Web.
  - Esta funcionalidade está disponível no Linux com a função **`dlopen`**
  - Exemplo: Para abrir uma função chamada **`libtest.so`**:
    - `dlopen("libtest.so", RTLD_LAZY)`
  - Para usar carga dinâmica inclua **`<dlfcn.h>`** no cabeçalho e compile com a opção **`-ldl`** (inclui a biblioteca **`libdl`**)
  - Mais informações em [1], cap. 2.3, pg. 43.

Exercício 2

a) Compile `test.c` e coloque-o em uma biblioteca estática **`libteststa.a`**
b) Compile `test.c` e coloque-o em uma biblioteca compartilhada **`libtestdyn.so`** .
c) Compile `app.c` e ligue-o com a biblioteca estática **`libteststa.a`**, gerando o executável **`appsta`** .
d) Execute **`appsta`** e verifique o código de retorno.
e) Compile `app.c` e ligue-o com a biblioteca compartilhada **`libtestdyn.so`**, gerando o executável **`appdyn`** .
f) Execute **`appdyn`** e verifique o código de retorno.
g) Utilize o comando **`ldd`** para verificar quais são as bibliotecas de que **`appdyn`** e **`appsta`** dependem. Quais são? Há diferença entre os dois casos?

Atenção: Coloque as respostas dos exercícios de 1 a 3 no Google Classroom.

Anotar os comandos utilizados em a, b, c, e.
Colocar as respostas de d, f, g.

Exercício 3

a) Modifique o código de retorno de `test.c` de **3** para **4**.

b) Compile test.c novamente e refaça **`libteststa.a`** e **`libtestdyn.so`**

(NÃO recompile* appdyn *e* appsta*!)

c) Execute **`appdyn`** e **`appsta`**. Os códigos de retorno são iguais? Por que?

Atenção: Coloque as respostas dos exercícios de 1 a 3 no Google Classroom.

Referências Bibliográficas

1. MITCHELL, Mark; OLDHAM, Jeffrey; SAMUEL, Alex; *Advanced Linux Programming*. New Riders Publishing: 2001. Cap. 2.