

Trabalho Prático II – ENTREGA: 06 de Dezembro de 2016

Implementação de um Sistema de Arquivos T2FS

1 Descrição Geral

O objetivo deste trabalho é a aplicação dos conceitos de sistemas operacionais na implementação de um Sistema de Arquivos que empregue alocação indexada combinada para a criação de arquivos e diretórios.

Esse Sistema de Arquivos será chamado, daqui para diante, de T2FS (*Task 2 – File System – Versão 2016.2*) e deverá ser implementado, OBRIGATORIAMENTE, na linguagem “C”, sem o uso de outras bibliotecas, com exceção da *libc*. Além disso, a implementação deverá executar na máquina virtual fornecida no Moodle.

O sistema de arquivos T2FS deverá ser disponibilizado na forma de um arquivo de biblioteca chamado *libt2fs.a*. Essa biblioteca fornecerá uma interface de programação através da qual programas de usuário e utilitários – escritos em C – poderão interagir com o sistema de arquivos.

A figura 1 ilustra os componentes deste trabalho. Notar a existência de três camadas de software. A camada superior é composta por programas de usuários, tais como os programas de teste (escritos pelo professor ou por vocês mesmos), e por programas utilitários do sistema.

A camada intermediária representa o Sistema de Arquivos T2FS. A implementação dessa camada é sua responsabilidade e o principal objetivo deste trabalho.

Por fim, a camada inferior, que representa o acesso ao disco, é implementada pela *apidisk*, que será fornecida junto com a especificação deste trabalho. A camada *apidisk* emula o *driver* de dispositivo do disco rígido e o próprio disco rígido. Essa camada é composta por um arquivo que simulará um disco formatado em T2FS, e por funções básicas de leitura e escrita de **setores lógicos** desse disco. As funções básicas de leitura e escrita simulam as solicitações enviadas ao *driver* de dispositivo (disco T2FS).

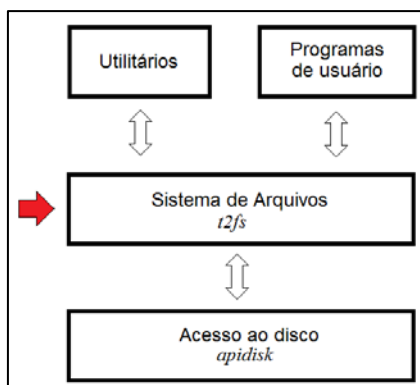


Figura 1 – Componentes principais do T2FS: aplicativos, sistema de arquivos e acesso ao disco.

2 Estrutura de um volume T2FS

O espaço disponível no disco formatado logicamente para T2FS (volume) está dividido em cinco áreas: superbloco, *bitmap* de blocos do disco livres e ocupados, *bitmap* de *i-nodes* livres e ocupados, *i-nodes* e, por fim, blocos de dados. A gerência do espaço em disco é feito por alocação indexada combinada e o diretório segue uma hierarquia em árvore. Um volume T2FS é formado por *S* setores (0 a *S*-1), onde os primeiros setores armazenam as estruturas gerenciais que definem e controlam a cartografia do disco T2FS, e os setores restantes são agrupados, *n* a *n*, formando um bloco de dados. Portanto, os blocos T2FS são formados por *n* setores contíguos e são numerados de 0 a *B*-1, onde *B* é o número total de blocos de dados. Esses

valores são fornecidos no superbloco (vide tabela 1) e a figura 2 mostra a estrutura do volume T2FS. As áreas do disco T2FS são detalhadas na sequência.

Tabela 1 – Descrição dos campos do superbloco

Posição relativa	Tamanho (bytes)	Nome	Valor	Descrição
0	4	<i>id</i>	"T2FS"	Identificação do sistema de arquivo. É formado pelas letras "T2FS".
4	2	<i>version</i>	0x7E02	Versão atual desse sistema de arquivos: (valor fixo 0x7E0=2016; 2=2º semestre).
6	2	<i>superblockSize</i>	1	Quantidade de setores ocupados pelo superbloco.
8	2	<i>freeBlocksBitmapSize</i>	1	Quantidade de setores usados para armazenar o <i>bitmap</i> de blocos de dados livres e ocupados.
10	2	<i>freeInodeBitmapSize</i>	1	Quantidade de setores usados para armazenar o <i>bitmap</i> de <i>i-nodes</i> livres e ocupados.
12	2	<i>inodeAreaSize</i>	125	Quantidade de setores usados para armazenar os <i>i-nodes</i> do sistema.
14	2	<i>blockSize</i>	16	Quantidade de setores que formam um bloco lógico.
16	4	<i>diskSize</i>	32768	Quantidade total de setores na partição T2FS. Inclui o superbloco, áreas de <i>bitmap</i> , área de <i>i-node</i> e blocos de dados
20 até o final (255)		<i>reservado</i>		Não usados

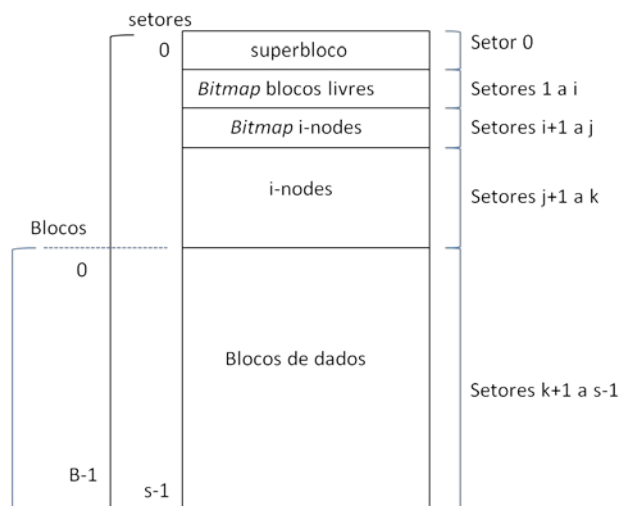
Superbloco: é a área de controle do sistema de arquivos. Essa área **ocupa o primeiro setor do disco**, conforme aparece na tabela 1. Todos os valores no superbloco são armazenados em um formato *little-endian* (a parte menos significativa do valor é armazenada no endereço mais baixo de memória).

Área de *bitmap* de blocos livres/ocupados: é formada pelo conjunto de n setores do disco, onde n é o valor definido em *freeBlocksBitmapSize* fornecido no superbloco. Cada *bit* nessa área corresponde a um bloco no disco e indica se o bloco está livre ou ocupado. (OBS.: As funções para alocar e liberar blocos via *bitmap* são fornecidas como parte integrante desta especificação. É obrigatório o uso dessas funções na implementação do T2FS). Se esse bit vale "1", o bloco está ocupado; se vale "0", está livre.

Área de *bitmap* de *i-nodes* livres/ocupados: é formada pelo conjunto de n setores do disco, onde n é o valor definido em *freeInodeBitmapSize* fornecido no superbloco. Cada *bit* nessa área corresponde a um *i-node* no disco e indica se o *i-node* está livre ou ocupado. (OBS.: As funções para alocar e liberar *i-nodes* via *bitmap* são fornecidas como parte integrante desta especificação. É obrigatório o uso dessas funções na implementação do T2FS). Se esse bit vale "1", o *i-node* está ocupado; se vale "0", está livre.

Área de *i-nodes*: conjunto de setores do disco onde estão armazenados os *i-nodes* do T2FS. O tamanho dessa área, em número de setores, é dado por *inodeAreaSize*, fornecido no superbloco.

Área para blocos de dados: área que inicia no primeiro setor após a área de *i-nodes* e se estende até o final do disco. É nessa área que estão os blocos de dados que formaram os arquivos. Se necessário, em função do tamanho de um arquivo, esses blocos poderão ser empregados como blocos de índices, ou seja, possuirão ponteiros para blocos de dados dos arquivos.



Cada bloco é formado por n setores (campo *sectorsPerBlock* do superbloco)

Figura 2 – Organização do disco lógico T2FS

2.1 Implementação de Diretórios no T2FS

O diretório T2FS segue uma organização em árvore, ou seja, dentro de um diretório é possível definir um subdiretório, e assim sucessivamente. Portanto, um diretório T2FS pode conter registros de:

- arquivos regulares;
- arquivos de diretórios (subdiretórios).

Os nomes dos arquivos no T2FS são dados apenas na forma absoluta, ou seja, o caminho do arquivo é sempre fornecido a partir do diretório raiz. O caractere de barra (" $/$ ") será utilizado na formação desses caminhos absolutos.

Cada arquivo (regular ou subdiretório) existente em um disco formatado em T2FS possui uma entrada (registro) em um diretório. Os diretórios são implementados por arquivos organizados internamente como uma lista linear de registros de tamanho fixo. A tabela 2 mostra a estrutura de um registro (estrutura *t2fs_record*), onde todos os valores numéricos são armazenados em formato *little-endian*.

Tabela 2 – Estrutura interna de uma entrada de diretório no T2FS (estrutura *t2fs_record*)

Posição relativa	Tamanho (bytes)	Nome	Descrição
0	1	<i>TypeVal</i>	Tipo da entrada. Indica se o registro é válido e, se for, o tipo do arquivo (regular ou diretório). <ul style="list-style-type: none">• 0x00, registro inválido (não associado a nenhum arquivo);• 0x01, arquivo regular;• 0x02, arquivo de diretório.• Outros valores, registro inválido (não associado a nenhum arquivo)
1	31	<i>name</i>	Nome do arquivo associado ao registro.
32	4	<i>blocksFileSize</i>	Tamanho do arquivo expresso em número de blocos.
36	4	<i>bytesFileSize</i>	Tamanho do arquivo expresso em número de bytes.
40	4	<i>inodeNumber</i>	Número do <i>i-node</i>
44 a 63	20		Não usados

2.2 Implementação de arquivos no T2FS

O T2FS é um sistema de arquivos que emprega o método de alocação indexada combinada e os arquivos podem ser do tipo regular ou diretório. Um arquivo regular é aquele que contém dados de um usuário podendo ser um arquivo texto (caracteres ASC II) ou binário. Sempre que um arquivo for criado, deve ser alocada uma entrada no diretório corrente e os campos da tabela 2 preenchidos de forma adequada.

À medida que um arquivo recebe dados, devem ser alocados os blocos lógicos necessários ao armazenamento desses dados. Sempre que um arquivo tiver sua quantidade de dados reduzida, os blocos lógicos que forem liberados devem ser tornados livres.

No caso da remoção de um arquivo, a entrada do diretório deve ser atualizada para “inválida” (campo *TypeVal* = 0x00) e todos os blocos lógicos empregados para o arquivo, tanto para dados quanto para índices, se for o caso, devem ser marcados como livre. Observe que ao remover uma entrada no diretório NÃO é necessário “compactar” o arquivo de diretório para eliminar essa entrada, basta marcá-la como inválida e reaproveitá-la em uma posterior criação de arquivos.

Os arquivos são descritos nos diretórios através de um registro chamado *entrada de diretório* (estrutura *t2fs_record*). Os campos da entrada de diretório (tabela 2) devem ser atualizados de forma a refletir corretamente o crescimento ou a redução do tamanho de um arquivo.

Os nomes simbólicos para os arquivos e diretórios do T2FS têm no máximo 32 caracteres. Os caracteres usados nos nomes simbólicos só podem ser letras, números e o caractere “.” (ponto). Os nomes simbólicos não necessitam ter extensão e são *case-sensitive*.

Por empregar alocação indexada combinada, é necessário fornecer uma lista de quais blocos constituem o arquivo. O T2FS possui essa informação na entrada do diretório através do atributo *i-node*. O formato do *i-node* T2FS é dado na tabela 3 e forma a estrutura *t2fs_inode*. Novamente, todos os valores estão em um formato *little-endian*.

Tabela 3 – Estrutura interna de um *i-node* T2FS (estrutura *t2fs_inode*)

Posição relativa	Tamanho (bytes)	Nome	Descrição
0	8	<i>dataPtr[2]</i>	Dois ponteiros diretos.
8	4	<i>singleIndPtr</i>	Ponteiro de indireção simples.
12	4	<i>doubleIndPtr</i>	Ponteiro de indireção dupla.

Assim, para arquivos com tamanho de até dois blocos lógicos, o campo *dataPtr* informa o endereço desses blocos lógicos. Para arquivos maiores que dois blocos lógicos, são empregados, além desses dois ponteiros diretos, os ponteiros de indireção simples ou dupla, os quais apontam para blocos lógicos que contém a continuação da lista de endereços de blocos lógicos que compõe o arquivo (bloco de índice). Os ponteiros não utilizados devem ser preenchidos com o valor de ponteiro inválido (INVALID_PTR), conforme definido no “*t2fs.h*”.

O arquivo correspondente ao diretório raiz é descrito pelo *i-node* ZERO.

2.3 Formato de um bloco de índices

Os blocos de índices são blocos lógicos do disco onde estão armazenados ponteiros para blocos de dados ou outros blocos de índices (no caso da indireção dupla), de um determinado arquivo. Cada ponteiro ocupa 4 (quatro) bytes e está armazenado em formato *little endian*.

A ocupação dos ponteiros no bloco de índices deverá ser feita em ordem, iniciando na primeira posição do bloco (posição zero). Os ponteiros não usados devem receber o valor INVALID_PTR, sendo essa a forma de identificar o término da lista de ponteiros.

Dessa forma, levando-se em consideração a estrutura de ponteiros e o disco, o maior tamanho de um arquivo será dado através da seguinte expressão:

$$Tamanho = blockSize \cdot \left(\left(\frac{blockSize}{4} \right)^2 + \left(\frac{blockSize}{4} \right) + 2 \right)$$

O final da lista de blocos lógicos é sinalizada por uma entrada com o valor INVALID_PTR.

3 Interface de Programação da T2FS (*libt2fs.a*)

Sua tarefa é implementar a biblioteca *libt2fs.a* que possibilitará o acesso aos arquivos regulares e de diretório do sistema de arquivos T2FS.

As funções a serem implementadas estão resumidas na tabela 4, onde são usados alguns tipos de dados e protótipos de função que estão definidos no arquivo *t2fs.h* fornecido junto com a especificação deste trabalho. A implementação de seu trabalho deve possuir TODAS AS FUNÇÕES especificadas aqui, mesmo que não tenham sido implementadas. Isso visa evitar erros de compilação com testes que utilizem todas as funções. REFORÇANDO: se você não implementar o corpo de uma função, crie a função conforme o *prototype* fornecido e, em seu corpo, coloque apenas o comando C *return* com um valor apropriado de acordo com o *prototype* da função.

A implementação do sistema de arquivos T2FS deve ser feita de tal forma que seja possível ter-se até 20 (vinte) arquivos abertos simultaneamente.

Tabela 4 – Interface de programação de aplicações – API - da *libt2fs*

Nome	Descrição
<code>int identify2 (char *name, int size)</code>	Informa a identificação dos desenvolvedores do T2FS.
<code>FILE2 create2 (char *filename)</code>	Função usada para criar um novo arquivo no disco.
<code>int delete2 (char *filename)</code>	Função usada para remover (apagar) um arquivo do disco.
<code>FILE2 open2 (char *filename)</code>	Função que abre um arquivo existente no disco.
<code>int close2 (FILE2 handle)</code>	Função usada para fechar um arquivo.
<code>int read2 (FILE2 handle, char *buffer, int size)</code>	Função usada para realizar a leitura de uma certa quantidade de bytes (<i>size</i>) de um arquivo.
<code>int write2 (FILE2 handle, char *buffer, int size)</code>	Função usada para realizar a escrita de uma certa quantidade de bytes (<i>size</i>) de um arquivo.
<code>int truncate2 (FILE2 handle)</code>	Função usada para truncar um arquivo. Remove do arquivo todos os bytes a partir da posição atual do contador de posição (<i>current pointer</i>), inclusive, até o seu final.
<code>int seek2 (FILE2 handle, unsigned int offset)</code>	Altera o contador de posição (<i>current pointer</i>) do arquivo.
<code>int mkdir2 (char *pathname)</code>	Função usada para criar um novo diretório.
<code>int rmdir2 (char *pathname)</code>	Função usada para remover (apagar) um diretório do disco.
<code>DIR2 opendir2 (char *pathname)</code>	Função que abre um diretório existente no disco.
<code>int readdir2 (DIR2 handle, DIRENT2 *dentry)</code>	Função usada para ler as entradas de um diretório.
<code>int closedir2 (DIR2 handle)</code>	Função usada para fechar um diretório.

4 Interface da *apidisk (libapidisk.o)*

Para fins deste trabalho, você receberá o binário *apidisk.o*, que realiza as operações de leitura e escrita do subsistema de E/S do disco usado pelo T2FS. Assim, o binário *apidisk.o* permitirá a leitura e a escrita dos setores lógicos do disco, que serão endereçados através de sua numeração sequencial a partir de zero. Os setores lógicos têm, sempre, 256 bytes. As funções dessa API estão descritas a seguir.

`int read_sector (unsigned int sector, char *buffer)`

Realiza a leitura do setor “*sector*” lógico do disco e coloca os bytes lidos no espaço de memória indicado pelo ponteiro “*buffer*”.

Retorna “0”, se a leitura foi realizada corretamente e um valor diferente de zero, caso tenha ocorrido algum erro.

`int write_sector (unsigned int sector, char *buffer)`

Realiza a escrita do conteúdo da memória indicada pelo ponteiro “*buffer*” no setor “*sector*” lógico do disco.

Retorna “0”, se a escrita foi bem sucedida; retorna um valor diferente de zero, caso tenha ocorrido algum erro.

Por questões de simplificação, o binário *apidisk.o*, que implementa as funções *read_sector()* e *write_sector()*, e o arquivo de inclusão *apidisk.h*, com os protótipos dessas funções, serão fornecidos pelo professor. Além disso, será fornecido um arquivo de dados para emulação do disco onde estará o sistema de arquivos T2FS.

Importante: a biblioteca *apidisk* considera que o arquivo que emula o disco virtual T2FS possui sempre o nome *t2fs_disk.dat* e esse arquivo deve estar localizado no mesmo diretório em que estão os programas executáveis que o utiliza.

5 Interface de *bitmap* (*bitmap2.o*)

Também como base para a realização do trabalho você receberá o binário *bitmap2.o*, que realiza as operações de alocação e liberação de blocos lógicos da área de dados. Esses blocos são usados para o conteúdo dos arquivos e para os blocos de índice. As funções da tabela 5 têm como primeiro parâmetro um *handle*, necessário para identificar a área de bitmap que se deseja manipular. Esse *handle* pode receber os valores *BITMAP_INODE* e *BITMAP_DADOS*, constantes definidas em *bitmap2.h*.

Tabela 5 – Interface das funções de bitmap

Nome	Descrição
<code>int getBitmap2 (</code> <code>int handle,</code> <code>int bitNumber)</code>	Retorna o valor do bit <i>bitNumber</i> da área de bitmap identificada por <i>handle</i> . Se houver erro retorna um número negativo. Se for bem sucedido retorna o valor do bit: “0” (zero) ou “1” (um)
<code>int setBitmap2 (</code> <code>int handle,</code> <code>int bitNumber,</code> <code>int bitValue)</code>	Seta o valor <i>bitValue</i> no bit <i>bitNumber</i> da área de bitmap identificada por “ <i>handle</i> ”. Se houver erro retorna um número negativo. Se for bem sucedido retorna “0” (zero)
<code>int searchBitmap2 (</code> <code>int handle,</code> <code>int bitValue)</code>	Procura, a partir do bit 0 da área de bitmaps identificada por <i>handle</i> , um bit com o valor <i>bitValue</i> . Se houver erro retorna um número negativo. Se for bem sucedido retorna o índice do bit encontrado. Se não houver bits disponíveis retorna um valor negativo, indicando erro.

Por questões de simplificação, o binário *bitmap2.o*, que implementa as funções de manipulação do *bitmap* de alocação de blocos, e o arquivo de inclusão *bitmap2.h*, com os protótipos dessas funções, serão fornecidos pelo professor.

6 Entregáveis: o que (não) deve ser entregue?

A entrega do trabalho será realizada através da submissão pelo **Moodle** de um arquivo *tar.gz*, cuja estrutura de diretórios deverá seguir, OBRIGATORIAMENTE, a mesma estrutura de diretórios do arquivo *t2fs.tar.gz* fornecido (conforme seção 7).

Utilize a estrutura de diretórios especificada para desenvolver seu trabalho. Assim, ao terminá-lo, basta gerar um novo arquivo *tar.gz*, conforme descrito no ANEXO II. Observe também o seguinte:

- **NÃO** inclua, no *tar.gz*, cópia da Máquina Virtual;
- **NÃO** serão aceitos outros formatos de arquivos, tais como *tgz*, *rar* ou *zip*.

Devem ser entregues:

- Todos os arquivos fonte (arquivos “.c” e “.h”) que formam a biblioteca “*libt2fs*”;
- Arquivo *makefile* para criar a “*libt2fs.a*”.

- O arquivo “*libt2fs.a*”

O *makefile* deverá ter, além das regras de compilação específicas, pelo menos duas regras:

- Regra “*all*”: responsável por gerar o arquivo *libt2fs.a*, no diretório *lib*.
- Regra “*clean*”: responsável por remover dos subdiretórios *bin* e *lib* todos os arquivos gerados pela compilação.

7 Arquivo *tar.gz*

Será fornecido pelo professor (disponível no **Moodle**) um arquivo *t2fs.tar.gz*, que deve ser descompactado conforme descrito no ANEXO II, de maneira a gerar em seu disco a estrutura de diretórios a ser utilizada, OBRIGATORIAMENTE, para a entrega do trabalho.

No diretório raiz (diretório *t2fs*) da estrutura de diretórios do arquivo *t2fs.tar.gz* está disponibilizado um arquivo *Makefile* de referência, que deve ser completado de maneira a gerar a biblioteca. Os subdiretórios do diretório *t2fs* são os seguintes:

/t2fs		
	<i>bin</i>	DIRETÓRIO: local onde serão postos os programas executáveis usados para testar a implementação, ou seja, os executáveis dos programas de teste.
	<i>include</i>	DIRETÓRIO: local onde são postos todos os arquivos “.h”. Nesse diretório deve estar o <i>t2fs.h</i> , o <i>apidisk.h</i> e o <i>bitmap2.h</i>
	<i>lib</i>	DIRETÓRIO: local onde será gerada a biblioteca <i>libt2fs.a</i> . (junção da <i>t2fs</i> com <i>apidisk.o</i> e <i>bitmap2.o</i>). Os binários <i>apidisk.o</i> e <i>bitmap2.o</i> também serão postos neste diretório.
	<i>src</i>	DIRETÓRIO: local onde são postos todos os arquivos “.c” (códigos fonte) usados na implementação do T2FS.
	<i>teste</i>	DIRETÓRIO: local onde são armazenados todos os arquivos de programas de teste (códigos fonte) usados para testar a implementação do T2FS.
	<i>makefile</i>	ARQUIVO: arquivo <i>makefile</i> com regras para gerar a <i>libt2fs</i> . Deve possuir uma regra <i>clean</i> , para limpar todos os arquivos gerados.

8 Avaliação

A avaliação do trabalho considerará as seguintes condições:

- Entrega do trabalho final dentro dos prazos estabelecidos;
- Obediência à especificação (formato e nome das funções);
- Compilação e geração da biblioteca sem erros ou *warnings*;
- Fornecimento de todos os arquivos solicitados conforme organização de diretórios fornecidos na seção 7;
- Execução correta dentro da máquina virtual *alunovm-sisop.oiva*.

Uma vez cumpridas as condições gerais, as implementações serão avaliadas e valorizadas da seguinte forma:

- 50,0 pontos de avaliação individual, referentes às **reuniões de acompanhamento**;
- 50,0 pontos de avaliação da implementação, referentes ao **funcionamento da biblioteca**.

Para avaliação durante as **reuniões de acompanhamento** serão observados os seguintes itens:

- a correta associação entre a implementação e os conceitos vistos em aula;

- a gestão do desenvolvimento;
- distribuição das atividades entre os componentes do grupo;
- planejamento e projeto inicial;
- evolução da implementação conforme o projeto, e realização de ajustes quando necessário.

A verificação do **funcionamento da biblioteca** será realizada pela observação dos seguintes itens:

- utilização de programas padronizados desenvolvidos pelo professor, para uso da biblioteca;
- clareza e organização do código;
- o uso de programação modular;
- correta utilização dos *makefiles*;
- correta utilização dos arquivos de header (sem código “C” dentro dos *includes*).

9 Data de entrega e avisos gerais – LEIA com MUITA ATENÇÃO!!!

- Faz parte da avaliação a obediência RÍGIDA aos padrões de entrega definidos na seção 6 (arquivos *tar.gz*, estrutura de diretórios, *makefile*, etc);
- O trabalho deverá ser desenvolvido em grupos com três componentes;
- As reuniões de projeto serão avaliadas e as notas alcançadas farão parte da nota final do trabalho. A ausência nas reuniões de projeto implicam em nota 0 (zero) naquela avaliação;
- O trabalho deverá ser apresentado no laboratório, no horário da aula, na data de **6 de dezembro de 2016**;
- Após a apresentação, TODOS os grupos deverão realizar o *upload* de um arquivo *tar.gz* com suas implementações, no **Moodle** da disciplina;
- Não haverá prorrogação de prazo, ou seja, NÃO há tolerância de atraso na entrega do trabalho. Não serão aceitos trabalhos entregues além do prazo estabelecido;
- Recomenda-se a troca de ideias entre os alunos. Entretanto, a identificação de cópias de trabalhos acarretará na aplicação do Código Disciplinar Discente e a tomada das medidas cabíveis para essa situação.

ANEXO A – Compilação e Ligação

1. Compilação de arquivo fonte para arquivo objeto

Para compilar um arquivo fonte (*arquivo.c*, por exemplo) e gerar um arquivo objeto (*arquivo.o*, por exemplo), pode-se usar a seguinte linha de comando:

```
gcc -c arquivo.c -Wall
```

Notar que a opção *-Wall* solicita ao compilador que apresente todas as mensagens de alerta (*warnings*) sobre possíveis erros de atribuição de valores a variáveis e incompatibilidade na quantidade ou no tipo de argumentos em chamadas de função.

2. Compilação de arquivo fonte DIRETAMENTE para arquivo executável

A compilação pode ser feita de maneira a gerar, diretamente, o código executável, sem gerar o código objeto correspondente. Para isso, pode-se usar a seguinte linha de comando:

```
gcc -o arquivo arquivo.c -Wall
```

3. Geração de uma biblioteca estática

Para gerar um arquivo de biblioteca estática do tipo “.a”, os arquivos fonte devem ser compilados, gerando-se arquivos objeto. Então, esses arquivos objeto serão agrupados na biblioteca. Por exemplo, para agrupar os arquivos “*arq1.o*” e “*arq2.o*”, obtidos através de compilação, pode-se usar a seguinte linha de comando:

```
ar crs libexemplo.a arq1.o arq2.o
```

Nesse exemplo está sendo gerada uma biblioteca de nome “*exemplo*”, que estará no arquivo *libexemplo.a*.

4. Utilização de uma biblioteca

Deseja-se utilizar uma biblioteca estática (chamar funções que compõem essa biblioteca) implementada no arquivo *libexemplo.a*. Essa biblioteca será usada por um programa de nome *myprog.c*.

Se a biblioteca estiver no mesmo diretório do programa, pode-se usar o seguinte comando:

```
gcc -o myprog myprog.c -lexemplo -Wall
```

Notar que, no exemplo, o programa foi compilado e ligado à biblioteca em um único passo, gerando um arquivo executável (arquivo *myprog*). Observar, ainda, que a opção *-l* indica o nome da biblioteca a ser ligada. Observe que o prefixo *lib* e o sufixo *.a* do arquivo não necessitam ser informados. Por isso, a menção apenas ao nome *exemplo*.

Caso a biblioteca esteja em um diretório diferente do programa, deve-se informar o caminho (*path* relativo ou absoluto) da biblioteca. Por exemplo, se a biblioteca está no diretório */user/lib*, caminho absoluto, pode-se usar o seguinte comando:

```
gcc -o myprog myprog.c -L/user/lib -lexemplo -Wall
```

A opção “*-L*” suporta caminhos relativos. Por exemplo, supondo que existam dois diretórios: *testes* e *lib*, que são subdiretórios do mesmo diretório pai. Então, caso a compilação esteja sendo realizada no diretório *testes* e a biblioteca desejada estiver no subdiretório *lib*, pode-se usar a opção *-L* com “*../lib*”. Usando o exemplo anterior com essa nova localização das bibliotecas, o comando ficaria da seguinte forma:

```
gcc -o myprog myprog.c -L../lib -lexemplo -Wall
```


ANEXO B – Compilação e Ligação

1. Desmembramento e descompactação de arquivo *.tar.gz*

O arquivo *.tar.gz* pode ser desmembrado e descompactado de maneira a gerar, em seu disco, a mesma estrutura de diretórios original dos arquivos que o compõe. Supondo que o arquivo *tar.gz* chame-se "*file.tar.gz*", deve ser utilizado o seguinte comando:

```
tar -zxvf file.tar.gz
```

2. Geração de arquivo *.tar.gz*

Uma estrutura de diretórios existente no disco pode ser completamente copiada e compactada para um arquivo *tar.gz*. Supondo que se deseja copiar o conteúdo do diretório de nome "*dir*", incluindo seus arquivos e subdiretórios, para um único arquivo *tar.gz* de nome "*file.tar.gz*", deve-se, a partir do diretório pai do diretório "*dir*", usar o seguinte comando:

```
tar -zcvf file.tar.gz dir
```

ANEXO C – Discos físicos

Setores físicos, setores lógicos e blocos lógicos

Os discos rígidos são compostos por uma controladora e uma parte mecânica, da qual fazem parte a mídia magnética (pratos) e o conjunto de braços e cabeçotes de leitura e escrita. O disco físico pode ser visto como uma estrutura tridimensional composta pela superfície do prato (cabeçote), por cilindros (trilhas concêntricas) que, por sua vez, são divididos em **setores físicos** com um número fixo de bytes.

A tarefa da controladora de disco é transformar a estrutura tridimensional (*Cylinder, Head, Sector* – CHS) em uma sequência linear de **setores lógicos** com o mesmo tamanho dos setores físicos. Esse procedimento é conhecido como *Linear Block Address* (LBA). Os setores lógicos são numerados de 0 até $S-1$, onde S é o número total de setores lógicos do disco e são agrupados, segundo o formato do sistema de arquivos, para formar os **blocos lógicos** (ou *cluster*, na terminologia da Microsoft).

Assim, na formatação física, os setores físicos contêm, dependendo da mídia, 256, 512, 1024 ou 2048 bytes e, por consequência, os setores lógicos também têm esse tamanho. No caso específico do T2F2, considera-se que os setores físicos têm 256 bytes. Ao se formatar logicamente o disco para o sistema de arquivos T2FS, os setores lógicos serão agrupados para formar os blocos lógicos do T2FS. Dessa forma, um bloco lógico T2FS é formado por uma sequência contígua de n setores lógicos. A figura C.1 ilustra esses conceitos de forma genérica.

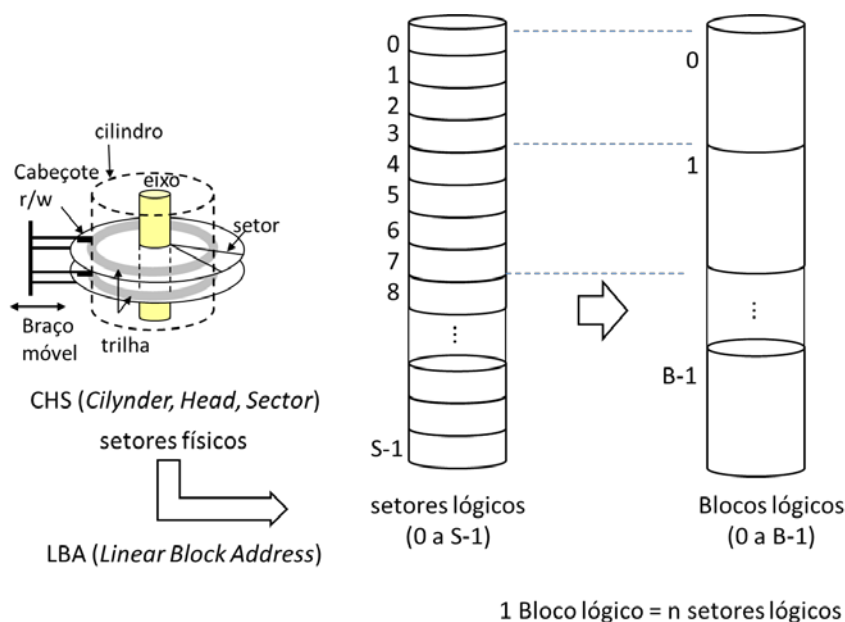


Figura C.1 – setores físicos, setores lógicos e blocos lógicos (diagrama genérico)