# Trabalho Final S.O.
# Sistemas de Arquivos

Gian Ricardo

June 2018

## 1 Objetivos

Ao realizar este projeto, você irá:

- Aprender sobre as estruturas de dado e implementação de um sistema de arquivo similar ao Unix.
- Aprender sobre recuperação de sistema de arquivo implementando uma varredura de bitmap de blocos livres.
- Desenvolver sua experiência em programação C++, utilizando estruturas e uniões extensivamente.
- Adquirir experiência ao desenvolver e empregar um regime rigoroso de testes.

## 2 Descrição

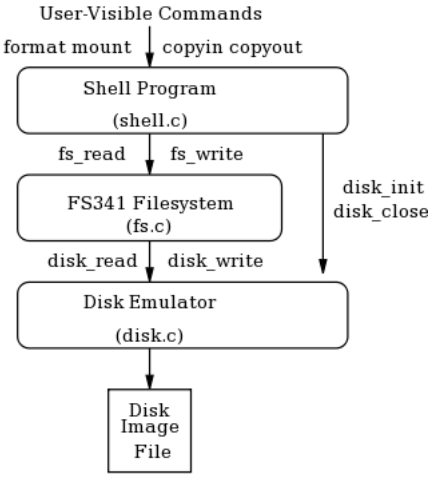
Neste projeto, você irá construir um sistema de arquivo do zero. Este sistema de arquivo se chama **SimpleFS** e é muito similar à camada de inodo (*inode*) Unix. Você irá começar com uma descrição detalhada de projeto do sistema de arquivos, e então escrever o código para implementá-lo.

Para permitir que você desenvolva e teste um novo sistema de arquivo sem a preocupação de danificar discos e sistemas de arquivo existentes, você irá construir o SimpleFS usando um **emulador** de disco. Um emulador de disco têm todas as capacidades de um disco real: ele carrega e armazena dados persistentes em blocos. Ele até reclama e dá crash como um disco real. A única diferença é que não há discos reais girando: o emulador apenas armazena o dado do disco como um arquivo em um sistema de arquivo existente.

Para testar a sua implementação do SimpleFS, nós iremos fornecer um shell que irá permitir você a criar, formatar, ler e escrever arquivos no seu sistema de arquivos. Nós também iremos fornecer vários exemplos de imagens de sistema de arquivo para que você não precise começar do zero.

### 2.1 Descrição do SimpleFS

O sistema SimpleFS tem 3 componentes principais: o shell, o sistema de arquivo em si, e o disco emulado. Seu trabalho é implementar o componente intermediário: o sistema de arquivos. A figura a seguir mostra como os componentes se relacionam entre si:

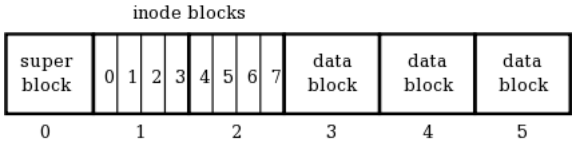


No nível mais alto o usuário fornece comandos digitados para um shell, instruindo-o a formatar ou montar o disco, e copiar dados para dentro ou fora do sistema do arquivo. O shell converte estes comandos digitados em operações de alto nível no sistema de arquivo, tal como *fs_format, fs_mount, fs_read e fs_write*. O sistema de arquivo é responsável por aceitar estas operações nos arquivos e convertê-los em operações de leitura e escrita de bloco simples no disco emulado, chamado de *disk_read e disk_write*. O disco emulado, por sua vez, armazena todos os seus dados em uma arquivo de imagem no sistema de arquivo.

Sua tarefa é concentrar-se no projeto e implementação do sistema de arquivo em si. Antes de entrar nos detalhes do sistema de emulação, falemos do sistema de arquivo em mais detalhes.

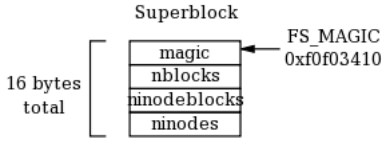
## 2.2 Projeto do sistema de arquivo SimpleFS

O SimpleFS tem o layout no disco a seguir. Ele assume que os blocos de disco têm um tamanho comum de 4kB. O primeiro bloco do disco é um "superbloco" que descreve o layout do resto do sistema de arquivo. Um certo número de blocos que segue o "superbloco" contém as estruturas de dados inodos. Tipicamente, dez por cento do número total de blocos de disco são usados como blocos de inodos. Os blocos remanescentes no sistema de arquivos são usados como blocos de dados comuns, e ocasionalmente como blocos ponteiros indiretos. Aqui está uma imagem de uma imagem SimpleFS muito pequena:




Vamos examinar cada um destes tipos de blocos em detalhe.

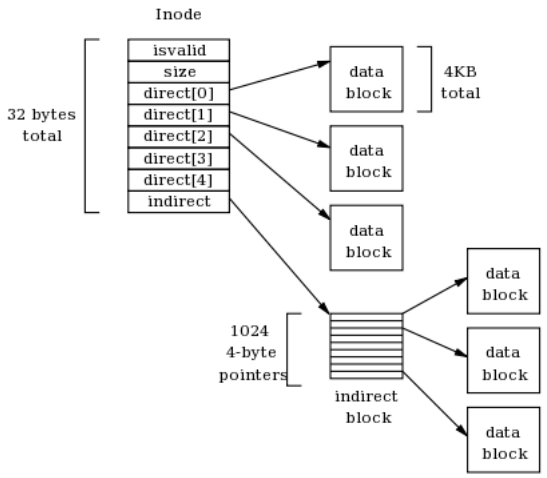
O superbloco descreve o layout do resto do sistema de arquivo:




Cada campo do superbloco é um inteiro de 4 bytes (32 bits). O primeiro campo é sempre o número "mágico" FS_MAGIC (0xf0f03410). A rotina de formatação coloca este número nos primeiros bytes do superbloco como um tipo de "assinatura" do sistema de arquivos. Quando o sistema de arquivos é montado, o SO procura por este número mágico. Se estiver correto, então assume-se que o disco contém um sistema de arquivos correto. Se algum outro número estiver presente, então a montagem falha, talvez porque o disco não esteja formatado ou contém algum outro tipo de dado.

Os campos remanescentes no superbloco descrevem o layout do sistema de arquivos. **nblocks** é o número total de bloco, que deveria ser o mesmo que o número de blocos no disco. **ninodeblocks** é o número de blocos reservados para armazenar inodos. **ninodes** é o número total de inodos nestes blocos. A rotina de formatação é responsável por escolher **ninodeblocks**: isto deve ser sempre 10 por cento de **nblocks**, arredondando pra cima. Note que a estrutura de dados do superbloco é pequena: apenas 16 bytes. O restante do bloco zero de disco é deixado sem ser usado.

Cada inodo se parece com o seguinte:



Cada campo do inodo é um campo de 4 bytes (32-bits). O campo **isvalid** é um se o inodo é válido (i.e se foi criado) e zero caso contrário. O campo **size** contem o tamanho lógico do dado do inode em bytes. Existem 5 ponteiros diretos para blocos de dados, e um ponteiro para um bloco de dados indireto. Neste contexto, "ponteiro" significa simplesmente o número de um bloco onde dado pode ser encontrado. Um valor de zero pode ser usado para indicar um ponteiro de bloco nulo. Cada inodo ocupa 32 bytes. Então existem 128 inodos em cada bloco de inodo de 4kb.

Note que um bloco de dados indireto é apenas um grande vetores de ponteiros para outros blocos de dados. Cada ponteiro é um inteiro de 4-bytes, e cada bloco é 4KB, então existem 1024 ponteiros por bloco. Os blocos de dados são simplesmente 4KB de dados "crus".

Você provavelmente deve ter notado que algo está faltando: um bitmap de blocos livres! Um sistema de arquivo real manteria um bitmap de blocos livres no disco, gravando um bit para cada bloco que estivesse disponível ou em uso. Este bitmap seria consultado e atualizado toda vez que o sistema de arquivo precisasse adicionar ou remover um bloco de dados de um inodo.

SimpleFS não tem espaço em disco para um bitmap de blocos livres. Então, SimpleFS requer que o projetista do sistema operacional (i.e. VOCÊ) mantenha um bitmap de blocos livres em memória. Isto é, deve haver um vetor de inteiros, um para cada bloco no disco, notando se o bloco está em uso ou disponível. Quando for necessário alocar um novo bloco para um arquivo, o sistema deve varrer através do vetor para localizar um bloco disponível. Quando um bloco for liberado, da mesma maneira, deve ser marcado no bitmap.

O que acontece quando memória é perdida? Suponha que o usuário faça algumas mudanças no sistema de arquivos SimpleFS, e então dê reboot no sistema. Sem um bitmap de blocos livres, o SimpleFS não consegue dizer quais blocos estão em uso e quais estão livres. Felizmente, esta informação pode ser recuperada lendo o disco. Cada vez que um sistema de arquivos SimpleFS é montado, o sistema deve construir um novo bitmap de blocos livres do zero varrendo por todos os inodos e gravando quais blocos estão em uso. (Isto é muito parecido com realizar um *fsck* toda vez que o sistema inicializa.)

O SimpleFS se parece muito com a camada de inodo do Unix. Cada "arquivo" é identificado por um inteiro chamado de "inúmero". O inúmero é simplesmente um índice para dentro do vetor de estruturas inodo que começa no bloco um. Quando um arquivo é criado, o SimpleFS, escolhe o primeiro inúmero disponível e o retorna para o usuário. Todas as referências subsequentes a este arquivo são feitas usando o inúmero. Usando o SimpleFS como uma base, você poderia facilmente adicionar outra camada de software que implementa nomes de arquivo e diretórios. No entanto, isto não fará parte do trabalho.

Agora que você sabe dos detalhes do sistema de arquivo a ser implementado, vamos discutir os detalhes técnicos do sistema de emulação.

## 2.3 Emulador de Disco

Nós fornecemos a você um emulador de disco no qual armazenar seu sistema de arquivo. Este "disco" é na verdade armazenado como um grande arquivo no sistema de arquivo, assim você pode salvar dados em uma imagem de disco e então recuperá-los depois. Além disto, nós vamos fornecer a você alguns exemplos de imagens de disco com os quais você pode experimentar para testar seu sistema de arquivo.

Tal como um disco real, o emulador apenas permite operações em blocos inteiros de disco de 4LB (DISK_BLOCK_SIZE). Você não pode ler ou escrever nenhuma unidade menor do que isto. O principal desafio de construir um sistema de arquivo é converter as operações requisitadas pelo usuário em quantidades arbitrárias de dados para operações em tamanhos fixos de blocos.

A interface do disco simulado é dado em *disk.h*:

```
#define DISK_BLOCK_SIZE 4096

int disk_init( const char *filename, int nblocks );
int disk_size();
void disk_read( int blocknum, char *data );
void disk_write( int blocknum, const char *data );
void disk_close();
```

Antes de realizar qualquer operação no disco, você deve chamar **disk_init** e especificar uma imagem de disco (real) para armazenar os dados de disco, é o número de blocos em um disco simulado. Se esta função é chamada em uma imagem de disco que já existe, o dado contido não será alterado. Quando você tiver terminado de usar o disco, chame **disk_close** para liberar o arquivo. Estas duas chamadas já são feitas para você no shell, então você não deve mudá-las.

Assim que o disco for inicializado, você pode chamar **disk_size()** para descobrir o numéro de blocos no disco. Como o nome sugere, **disk_read() e disk_write()** lê e escreve um bloco de dada no disco. Note que o primeiro argumento é um número de bloco, para que a chamada para **disk_read(0,data)** leia os primeiros 4KB de dado no disco, e **disk_read(1,data)** leia o próximo bloco de dado de 4KB no disco. Toda vez que você invocar um read ou write, você deve se assegurar de que **data** aponte para 4KB completos de memória.

Note que o disco tem algumas conveniências de programação que um disco real não tem. O disco real é um tanto "mimado" – se você enviar comandos inválidos, ele provavelmente irá dar crash no sistema ou se comportar de maneiras estranhas. Este disco simulado é mais "útil". Se você enviar um comando inválido, ele irá interromper o programa com uma mensagem de erro. Por exemplo, se você tentar ler ou escrever um bloco de disco que não existe, você irá receber esta mensagem de erro:

```
ERROR: blocknum (592) is too big!
Abort (core dumped)
```

## 2.4 Interface do Sistema de Arquivo

Usando o disco simulado existente, você irá construir um sistema de arquivos funcional. Note que nós já construímos a interface para o sistema de arquivos e fornecemos o código-base. A interface é dada em *fs.h*:

```
int fs_format();
void fs_debug();
int fs_mount();

int fs_create();
int fs_delete( int inumber );

int fs_getsize( int inumber );

int fs_read( int inumber, char *data, int length, int offset );
int fs_write( int inumber, const char *data, int length, int offset );
```

As várias funções devem funcionar da seguinte maneira:

- fs_debug:Varre um sistema de arquivo montado e reporta como os inodos e os blocos estão organizados. Se você conseguir escrever esta função, você já ganhou metade da batalha! Assim que você conseguir varrer e reportar as estruturas do sistema de arquivos, o resto é fácil. Sua saída do *fs_debug* deve ser similar ao seguinte:

  ```
  superblock:
          magic number is valid
          1010 blocks on disk
          101 blocks for inodes
          12928 inodes total
  inode 3:
          size: 45 bytes
          direct blocks: 103 194
  inode 5:
          size 81929 bytes
          direct blocks: 105 109
          indirect block: 210
          indirect data blocks: 211 212 213 214 ...
  ```

- fs_format – Cria um novo sistema de arquivos no disco, destruindo qualquer dado que estiver presente. Reserva dez por cento dos blocos para inodos, libera a tabela de inodos, e escreve o superbloco. Retorna um para sucesso, zero caso contrário. Note que formatar o sistema de arquivo não faz com que ele seja montado. Também, uma tentativa de formatar um disco que já foi montado não deve fazer nada e retornar falha.

- fs_mount - Examina o disco para um sistema de arquivo. Se um está presente, lê o superbloco, constroi um bitmap de blocos livres, e prepara o sistema de arquivo para uso. Retorna um em caso de sucesso, zero caso contrário. Note que uma montagem bem-sucedida é um pré-requisito para as outras chamadas.
- fs_create – Cria um novo inodo de comprimento zero. Em caso de sucesso, retorna o inúmero (positivo). Em caso de falha, retorna zero. (Note que isto implica que zero não pode ser um inúmero válido.)
- fs_delete – Deleta o inodo indicado pelo inúmero. Libera todo o dado e blocos indiretos atribuídos a este inodo e os retorna ao mapa de blocos livres. Em caso de sucesso, retorna um. Em caso de falha, returna 0.
- fs_getsize – Retorna o tamanho lógico do inodo especificado, em bytes. Note que zero é um tamanho lógico válido para um inodo! Em caso de falha, retorna -1.
- fs_read – Lê dado de um inodo válido. Copia "length" bytes do inodo para dentro do ponteiro "data", começando em "offset" no inodo. Retorna o número total de bytes lidos. O Número de bytes efetivamente lidos pode ser menos que o número de bytes requisitados, caso o fim do inodo seja alcançado. Se o inúmero dado for inválido, ou algum outro erro for encontrado, retorna 0.
- fs_write – Escreve dado para um inodo válido. Copia "length" bytes do ponteiro "data" para o inodo começando em "offset" bytes. Aloca quaisquer blocos diretos e indiretos no processo. Retorna o número de bytes efetivamente escritos. O número de bytes efetivamente escritos pode ser menor que o número de bytes requisitados, caso o disco se torne cheio. Se o inúmero dado for inválido, ou qualquer outro erro for encontrado, retorna 0.

É bem provável que o módulo do sistema de arquivo irá precisar de um número de variáveis globais para manter o controle do atual sistema de arquivos montado. Por exemplo, você irá certamente precisar de uma variável global para manter o controle do bitmap de blocos livres atual, e talvez também de outros itens. Você pode ter ouvido de outros lugares que variáveis globais é algo "ruim". No contexto de sistemas operacionais, elas são comuns e relativamente normais. Cuide para que não ocorra condições de corrida.

## 2.5 Interface Shell

Nós fornecemos para você um shell simples que será usado para testar seu sistema de arquivo e o disco simulado. Na correção do trabalho, nós iremos usar o shell para testar o seu código, então certifique-se de testá-lo extensivamente. Para usar o shell, rode *simplefs* com o nome de uma imagem de disco, e o número de bloco desta imagem. Por exemplo, para usar o exemplo *image.5* dado abaixo, rode:

```
% ./simplefs image.5 5
```

Ou, para iniciar com uma imagem de disco nova, apenas dê um novo nome de arquivo e número de blocos:

```
% ./simplefs mydisk 25
```

Assim que o shell iniciar, você pode usar o comando help para listar os comandos disponíveis:

```
simplefs> help
Commands are:
        format
        mount
        debug
        create
        delete < inode >
        cat < inode >
        copyin < file >< inode >
        copyout< inode >< file >
        help
        quit
        exit
```

A maioria dos comandos correspondem de uma maneira muito próxima à interface de sistema de arquivo. Por exemplo, *format, mount, debug, create e delete* chamam as funções correspondentes no sistema de arquivo.

Certifique-se de que você chame essas funções em uma ordem sensata. Um sistema de arquivos deve ser formatados uma vez antes que possa ser usado. Igualmente, deve ser montado antes de ser escrito ou lido.

Os comandos complexos são *cat, copyin, e copyout cat* lê um arquivo inteiro do sistema de arquivo e mostra no console, assim como o comando Unix de mesmo nome. *copyin e copyout* copiam um arquivo do sistema de arquivos Unix local para dentro do seu sistema de arquivos emulado. Por exemplo, para copiar o arquivo de dicionário para o inodo 10 no seu sistema de arquivo, faça o seguinte:

```
simplefs> copyin /usr/share/dict/words 10
```

Note que estes três comandos executam fazendo um grande número de chamadas para *fs_read e fs_write* para cada arquivo a ser copiado.

# 3 Começando

Baixe o código-fonte e construa-o com make.

Aqui estão algumas imagens de disco de exemplo para você começar. O nome de cada imagem de disco te informa quantos blocos existem em cada imagem. Cada imagem contém alguns documentos e arquivos familiares. Assim que você conseguir ler o que está nessas imagens, você deve prosseguir, escrevendo e modificando-os.

- image.5
- image.20
- image.200

*shell e disk* são dados e completamente implementados, e *fs* é um "esqueleto" esperando pelo seu trabalho. Nós fornecemos as primeiras linhas de *fs_debug* para te dar a idéia inicial. Para começar, contrua e execute *simplefs* e chame *debug* para ler e ter como saída o superbloco:

```
% make
% ./simplefs image.5 5
 simplefs> debug
superblock:
        5 blocks
        1 inode blocks
        128 inodes
```

# 4 Notas de Implementação

Sua tarefa é implementar o SimpleFS como descrito acima, preenchendo a implementação de *fs.c*. Você não precisa alterar quaisquer outros módulos de código. Nós já criamos algumas estruturas de dado de amostra para você começar. Estas podem ser encontradas em *fs.c*. Para começar, nós definimos um número de macros para constantes comuns que você irá usar. A maioria destas devem ser auto-explicativas:

```
#define DISK_BLOCK_SIZE 4096
#define FS_MAGIC 0xf0f03410
#define INODES_PER_BLOCK 128
#define POINTERS_PER_INODE 5
#define POINTERS_PER_BLOCK 1024
```

Note que *POINTS_PER_INODE* é o número de ponteiros diretos em cada estrutura de inodo, enquanto que POINTERS_PER_BLOCK é o número de ponteiros a ser encontrados em um bloco indireto. Repense esses valores como constantes.

O superbloco e estruturas inodo são facilmente traduzidas das figuras acima:

```
struct fs_superblock {
        int magic;
        int nblocks;
        int ninodeblocks;
        int ninodes;
};

struct fs_inode {
        int isvalid;
        int size;
        int direct[POINTERS_PER_INODE];
        int indirect;
};
```

Atente-se que vários inodos podem caber em um bloco de disco. Um pedaço de 4KB de memória correspondente a um bloco indireto pareceria com o seguinte:

```
int pointers[1024];
```

Finalmente, cada bloco de dados é apenas dados binários "crus" usados para armazenad o conteúdo parcial de um arquivo. Um bloco de dados pode ser especificado em C como simplesmente um vetor de 4096 bytes:

```
char data[4096];
```

Um bloco de disco "cru" pode ser usado para representar 4 tipos de dado: um superbloco, um bloco de 128 inodos, um ponteiro indireto de bloco, ou um simples bloco de dados. Isto implica em um problema de engenharia de software: como transformamos o dado "cru" retornado por *disk_read* em cada um dos 4 tipos de blocos de dados?

C fornece um pedaço de sintaxe para exatamente este problema. Nós podemos declarar uma *union* de cada um dos nossos 4 tipos de dados. Uma *union* se parece com uma *struct*, mas força todos os seus elementos a compartilhar o mesmo espaço de memória. Você pode pensar como uma *union* de vários tipos diferentes, todas sobrepostas em cima das outras.

Aqui está um exemplo. Nós criamos um *union fs_block* que representa as quatro maneiras diferentes de interpretar dados de disco crus:

```
union fs_block {
        struct fs_superblock super;
        struct fs_inode inodes[128];
        int pointers[1024];
        char data[4096];
};
```

Note que o tamanho de uma união *fs_block* será exatamente 4KB: o tamanho do maior membro da união. Para declarar uma variável do tipo: union *fs_block*:

```
union fs_block block;
```

Agora, nós podemos usar *disk_read* para carregar o dado cru do bloco zero. Nós damos ao *disk_read* a variável *block.data*, que se assemelha a um array de caracteres:

```
disk_read(0,block.data);
```

Mas, nós podemos interpretar este dado como se fosse uma *struct superblock* acessando a parte super da união. Por exemplo, para extrair o número mágico do super bloco, nós podemos fazer isto:

```
x = block.super.magic;
```

Por outro lado, suponha que queiramos carregar o bloco de disco 59, assumindo que é um bloco indireto, e então examinas o quarto ponteiro. Novamente, nós usaríamos *disk_read* para carregar o dado cru:

```
disk_read(59,block.data);
```

Mas então usaríamos a parte *pointer* da união da seguinte maneira:

```
x = block.pointer[4];
```

A união oferece uma maneira conveniente de visualizar o mesmo dado em múltiplas perspectivas. Quando nós carregamos dado do disco, é apenas um pedaço de dados cru de 4KB (block.data). Mas, assim que carregado, a camada do sistema de arquivo sabe que este dado tem alguma estrutura. A camada do sistema de arquivo pode visualizar o mesmo dado sob outra perspectiva escolhendo outro campo na união.(block.super)

# 5 Conselhos Gerais

1. Implemente as funções aproximadamente em ordem. Nós intencionalmente apresentamos as funções do sistema de arquivos em ordem de dificuldade. Implemente debug, format, e mount primeiro. Assegure-se que você seja capaz de acessar as imagens de disco de exemplo fornecidos. Então, realize a criação e deleção de inodos sem se preocupar com dados de blocos. Implemente a leitura e teste novamente com as imagens do disco. Se todo o resto estiver funcionando, então tente fs_write.

2. Dividir para conquistar. Trabalhe duro para separar ações comuns em funções simples. Isto irá simplificar seu código dramaticamente. Por exemplo, você irá frequentemente precisar carregar e salvar estruturas de inodos individuais por número. Isto envolve um pouco de computação para transformar um inúmero em um número de bloco, e assim por diante. Então, faça duas pequenas funções para fazer exatamente isto:

   ```
   void inode_load( int inumber, struct fs_inode *inode ) { ... }
   void inode_save( int inumber, struct fs_inode *inode ) { ... }
   ```

   Agora, sempre que você precisar carregar ou salbar uma estrutura de inodo, chame estas funções. Sempre que você se encontrar escrevendo trechos de códigos muito similares repetidamente, separe-os em funções menores.

3. Teste condições de contorno. Nós vamos certamente testar seu código analisando as exceções. Assegure-se de testar e consertar condições de contorno antes de entregar o trabalho. Por exemplo, o que acontece se *fs_create* descobre que uma tabela de inodo está cheia? Deve, de maneira clara, retornar um código de erro. Certamente não deve dar crash no programa ou mutilar o disco! Pense criticamente sobre outras possíveis condições de contorno tal como o fim de um arquivo ou um disco cheio.

4. Não se preocupe com performance. Você será avaliado pela exatidão, não performance. Aliás, ao longo deste trabalho, você descobrirá que um simples acesso de arquivo pode facilmente resultar em dezenas ou centenas de acessos individuais ao disco. Entenda porque isto ocorre, mas não se preocupe com a otimização.

# 6 Entregando o Trabalho

Este trabalho deve ser entregue pelo Moodle. Entregas após a data limite não serão aceitos!

Para entregar sua atividade, comprima todos os arquivos .cpp e .h, juntamente com um Makefile em um único arquivo e submeta no moodle. Certifique-se de que você está usando */usr/bin/g++* como compilador. Você pode verificar se está utilizando o compilador correto com o comando *which g++*. Se você estiver vendo alguma outra coisa, você está utilizando o compilador errado.

# 7 Crédito Extra

Adicione um comando shell para defrag (desfragmentar) uma imagem montada. O comando defrag deve re-arranjar o conteúdo para que fique contíguo no disco para todos os objetos assim como o empacotamento no nós iniciais (excetuando o inodo inválido 0). Crie um arquivo chamado EC.txt que descreva sua abordagem e forneça um example *fs_debug* antes e depois de executar defrag.

1. 1 ponto (Opção 1): Escreva uma implementação correta de defrag
2. 2,5 pontos (Opção 2): Escreva uma implementação correta de defrag que mantenha no máximo apenas 2 blocos de dado na memória (número ilimitado de inodos)

   Você pode realizar uma das duas opções: Opção 1 ou 2.

Adaptado de material do Prof. Douglas Thain.

# 8 Auxílios

## 8.1 Sáida esperada da função debug para o disco.20

```
fs-shell> debug
superblock:
        20 blocks
        3 inode blocks
        384 inodes
inode 2:
        size: 27160 bytes
        direct blocks: 4 5 6 7 8
        indirect block: 9
        indirect data blocks: 13 14
inode 3:
        size: 9546 bytes
        direct blocks: 10 11 12
```