UNIVERSIDADE FEDERAL DE GOIÁS – UFG
INSTITUTO DE INFORMÁTICA – INF

Arthur de Oliveira Barbosa Lacerda
Murillo Rodrigues de Paula

Relatório do desenvolvimento do sistema de arquivos FAT16

Linguagens de Programação
Prof. Dr. Bruno Oliveira Silvestre

Goiânia, 2017

SUMÁRIO

1. CONVENÇÕES DO CÓDIGO

- Todas as variáveis utilizadas foram nomeadas na língua inglesa e apresentam formato em camel case.
  - Ex: variableNameExample
- Toda função foi nomeada na língua inglesa, utilizando apenas letras minusculas e tem o caracter “_” como separador.
  - Ex: function_example
- Todo o comentário do código também está na língua inglesa.
  - Padrão de comentário de funções:

```
/**
 * Description: This is the function description
 * ============================================
 * Return
 * @returnname: What the return means.
 * ============================================
 * Parameters
 * @param1name: What parameter 1 mean.
 * @param2name: What parameter 2 mean.
**/
```

  - Padrão de comentário dentro das funções:

```
/* This is a single line comment */

/* This is the format of a multiple
 * line comment */
```

- Tamanho 2 de tabulação.

2. ESTRUTURAS E TIPOLOGIAS UTILIZADAS

Por questão de compatibilidade e uniformidade com a FAT16, todas os tipos são unsigned por padrão.
O tipo BYTE é definido como uma quantidade de 8 bits sem sinal.
O tipo WORD é definido como uma quantidade de 16 bits sem sinal.
O tipo DWORD é definido como uma quantidade de 32 bits sem sinal.

Segue abaixo às estuturas e definições utilizadas:

```
typedef uint8_t BYTE;
typedef uint16_t WORD;
typedef uint32_t DWORD;

typedef struct {
  BYTE BS_jmpBoot[3];
  BYTE BS_OEMName[8];
  WORD BPB_BytsPerSec;
```

```
  BYTE BPB_SecPerClus;
  WORD BPB_RsvdSecCnt;
  BYTE BPB_NumFATS;
  WORD BPB_RootEntCnt;
  WORD BPB_TotSec16;
  BYTE BPB_Media;
  WORD BPB_FATSz16;
  WORD BPB_SecPerTrk;
  WORD BPB_NumHeads;
  DWORD BPB_HiddSec;
  DWORD BPB_TotSec32;
  BYTE BS_DrvNum;
  BYTE BS_Reserved1;
  BYTE BS_BootSig;
  DWORD BS_VollID;
  BYTE BS_VollLab[11];
  BYTE BS_FilSysType[8];
  BYTE Reserved[448];
  WORD Signature_word;
} __attribute__ ((packed)) BPB_BS;

typedef struct {
  BYTE DIR_Name[11];
  BYTE DIR_Attr;
  BYTE DIR_NTRes;
  BYTE DIR_CrtTimeTenth;
  WORD DIR_CrtTime;
  WORD DIR_CrtDate;
  WORD DIR_LstAccDate;
  WORD DIR_FstClusHI;
  WORD DIR_WrtTime;
  WORD DIR_WrtDate;
  WORD DIR_FstClusLO;
  DWORD DIR_FileSize;
} __attribute__ ((packed)) DIR_ENTRY;

typedef struct {
  DWORD FirstRootDirSecNum;
  DWORD FirstDataSector;
  BYTE *Fat;
  BPB_BS Bpb;
} VOLUME;
```

3. FUNÇÃO sector_read

A função *sector_read* à ser implementada em *sector.c* simplemente deveria acessar o posicionamento da FAT16 referente ao parâmetro secnum (número do setor) e efetuar a leitura.

Para isso foi necessário apenas utilizar a função *fseek* de tal forma que o descritor, à partir do SEEK_SET (Ponto inicial da FAT16), desse um salto de 512*secnum, e logo após a leitura dos próximos 512 bytes seria efetuada e armazenada no buffer.

Segue abaixo a implementação de *sector_read*.

```
void sector_read(FILE *fd, unsigned int secnum, void *buffer)
{
  fseek(fd, BYTES_PER_SECTOR * secnum, SEEK_SET);
  fread(buffer, BYTES_PER_SECTOR, 1, fd);
}
```

4. IMPLEMENTAÇÃO FAT16

Primeiramente escrevemos o código run_fat16.c afim de entender e efetuar o funcionamento da FAT16.

O programa recebe como parâmetro a imagem da FAT16 e o caminho à ser percorrido, e é apresentado na saída se o caminho foi ou não percorrido com sucesso na imagem.

Para compilar e executar o programa, deve-se utilizar os comandos à seguir:

```
gcc run_fat16.c sector.c -o run_fat16
./run_fat16 <Imagem FAT16> <Diretório>
```

4.1. Funções utilizadas

4.1.1 *path_treatment*

retorno:

char** pathFormatted: Vetor de strings com cada string referente à um arquivo do caminho em uma posição, na devida formatação da FAT16.

parâmetros:

1. char* pathInput : String dada de entrada ao programa com o caminho de diretórios.
2. int* pathSz: Endereço da variável que armazenará a quantidade de arquivos presentes no caminho.

Descrição:

A função trata a string dada de entrada adaptando para o formato FAT16.

O caminho dado de entrada (pathInput) é dividido em um vetor de strings (path) que eram anteriormente delimitadas pelo token “/”, a função *strtok* da biblioteca *string.h* é utilizada para este fim.

Após esse procedimento, é criado um vetor de strings (pathFormatted), cada uma com tamanho 11, que armazenará o nome de cada arquivo no formato da FAT16.

O tratamento das strings então é realizada, de forma que a informação passada de path seja transferida para pathFormatted com as devidas alterações, sendo estas que

aceite apenas os caracteres aceitos pela especificação, troque caracteres em caixa baixa por caixa alta, que os 8 primeiros espaços das string sejam referentes ao nome do arquivo e os três últimos à extensão e qualquer espaço inutilizado é preenchido com o caracter de espaço “ “, também é tratado o caso em que os arquivos possam ser “.” ou “..”.

4.1.2. *fat16_init*

retorno:

VOLUME* Vol: Estrutura com dados essenciais da FAT16

parâmetros:

1. FILE* fd: Descritor de arquivo.

descrição:

A leitura do primeiro setor é efetuada, que é referente ao setor BPB, e o grava no volume (Vol->Bpb).

A posição do primeiro setor do diretório raiz (Vol->FirstRootDirSecNum) é calculado, tal calculo é feito baseado no salto da área reservada (BPB_RsvdSecCnt), acrescido do tamanho de uma FAT (BPB_FATSz16) multiplicado pela quantidade de FATs (BPB_NumFATS).

O número de setores do diretório raiz então é calculado baseado na quantidade de entradas do diretório raiz. Com esse valor em memória, calcula-se então a posição do primeiro setor da região de dados (Vol->FirstDataSector), que é dada pelo primeiro setor do diretório raiz acrescido do número de setores do diretório raiz.

4.1.3. find_root

retorno:

void

parâmetros:

1. FILE* fd: Descritor do arquivo.
2. VOLUME Vol: Estrutura com dados essenciais da FAT16.
3. DIR_ENTRY Root: Variável que armazenará entradas de diretório situadas no diretório raiz.
4. char **path: Vetor de arquivos do caminho à ser percorrido.
5. int pathDepth: Profundidade (ou índice) do caminho.
6. int pathSize: Tamanho total do caminho.

descrição:

A função percorre as entradas de diretório raiz buscando o arquivo da atual profundidade do caminho.

Com a função *sector_read*, lê-se todas às entradas de diretórios da root, até que encontre o arquivo especificado ou até que chegue em seu fim.

A comparação de string é feita caracter por caracter entre o path e Root.DIR_Name.

Se o arquivo for encontrado e for o último do caminho, então ele é apresentado, se for encontrado e não for o último, então a função *find_subdir* é chamada incrementando a profundidade do caminho.

4.1.4. fat_entry_by_cluster

retorno:

void

parâmetros:

1. FILE* fd: Descritor do arquivo.
2. VOLUME* Vol: Estrutura com dados essenciais da FAT16.
3. WORD ClusterN: Cluster em que será usado como base para obter a entrada da FAT.

Descrição:

Dado um cluster N, é determinada a entrada da FAT.

Primeiramente define-se o FATOffset da FAT como ClusterN*2, já que uma entrada da fat tem 2 bytes (16bits).

Calcula-se então o número de setor da FAT em que a leitura deve ser realizada, saltando a área reservada acrescida do número de setores que o offset da fat passa (FATOFFSET / Vol->Bpb.BPB_BytsPerSec).

Calcula-se então o offset de entrada, de acordo com o resto da divisão do FATOffset com Vol->Bpb.BPB_BytsPerSec.

Lê-se o setor correspondente à ao numero do setor calculado e retorna o valor encontrado no offset de entrada.

4.1.5. find_subdir

retorno:

void

parâmetros:

1. FILE* fd: Descritor do arquivo.
2. VOLUME Vol: Estrutura com dados essenciais da FAT16.
3. DIR_ENTRY Dir Variável que armazenará entradas de diretório situadas no subdiretório.
4. char** path
5. int pathSize: Vetor de arquivos do caminho à ser percorrido.

6. int pathDepth: Profundidade (ou índice) do caminho.
7. int rootDepth: Profundidade atual do caminho em relação ao diretório raiz.

descrição:

A função percorre as entradas do subdiretório buscando o arquivo da atual profundidade do caminho.

Com a função *fat_entry_by_cluster* determina-se a entrada da FAT para qual o subdiretório terá continuidade na busca. Caso o cluster seja o último então terá valor 0xFFFF.

Com a função *sector_read*, lê-se todas às entradas do subdiretório, até que encontre o arquivo especificado ou até que chegue ao fim do cluster.

Se chegar ao fim do cluster, o valor do cluster atual passa a ser a entrada da FAT anterior, e então se recalcula a próxima entrada da FAT, reinicializando o processo e dando continuidade à leitura do subdiretório.

A comparação de string é feita caracter por caracter entre o path e Dir.DIR_Name.

Se o arquivo for encontrado e for o último do caminho, então ele é apresentado, se for encontrado e não for o último, então a função *find_subdir* é chamada recursivamente incrementando a profundidade do caminho, além disso, se o arquivo da profundidade atual for ".", a profundidade em relação ao diretório raiz não muda, e se for ".." a profundidade em relação ao diretório raiz diminui. Se chegar ao ponto de que a profundidade em relação ao diretório raiz chegar a zero, então em vez de chamar a função *find_subdir*, chamaremos *find_root* para retornar ao diretório raiz e continuar a percorrer o caminho..

4.2. Funcionamento

Na função main do programa desenvolvido, primeiramente abre-se a imagem passada como primeio parâmetro, e logo após, trata-se a string passada como segundo parâmetro com a função *path_treatment*, inicializa-se então o Volume com a função *fat16_init*, afim de armazenar o BPB e determinar os setores de início do diretório raiz e da área de dados, e então chama-se a função de busca *find_root* que iniciará a busca pela primeira profundiade do caminho, que chamará *find_subdir* ao encontrar o primeiro aquivo, e entrará em recursão até encontrar o último (ou voltará a *find_root* caso a profundidade relativa ao diretório raiz seja zero. Ao encontrar o arquivo final do caminho, ele é mostrado na tela, se o arquivo em alguma profundidade não for encontrado ao varrer o subdiretório (ou diretório raiz) atual, então uma mensagem é mostrada na tela informando que não há tal arquivo.

Abordando em um nível estrutural, primeiramente é lido o primeiro setor da FAT16, e então salta-se para o primeiro setor diretório raiz, e então busca-se entrada por entrada, àquela que seja correspondente ao arquivo encontrado, então move-se para a região de dados, e busca a entrada do próximo cluster na região da fat, e assim segue-se a leitura, fazendo consultas entra a região de dados e a fat, e ocasionalmente dependendo do caminho, poderá voltar à região do diretório raíz.

5. FUSE

O FUSE (Filesystem in USErspace) é uma ferramenta que permite ao usuário criar seus próprios sistemas de arquivo sem que seja necessário editar o modo kernel.

Para nosso trabalho, implementamos o sistema de arquivo FAT16 no FUSE, e para isso, é necessário implementar funções que o FUSE requisita ao usuário ao fazer a comunicação do nível usuário ao kernel.

5.1. Funções necessárias do FUSE

init inicializa o sistema de arquivos

```
void*(* fuse_operations::init)(struct fuse_conn_info *conn, struct fuse_config *cfg)
```

getattr é responsável por obter os atributos do arquivo.

```
int(* fuse_operations::getattr)(const char *, struct stat *, struct fuse_file_info *fi)
```

opendir abre o um diretório

```
int(* fuse_operations::opendir)(const char *, struct fuse_file_info *)
```

readdir lê um diretório

```
int(* fuse_operations::readdir)(const char *, void *, fuse_fill_dir_t, off_t, struct fuse_file_info *, enum fuse_readdir_flags)
```

releasedir libera um diretório aberto

```
int(* fuse_operations::releasedir)(const char *, struct fuse_file_info *)
```

open é a operação de abertura de um arquivo

```
int(* fuse_operations::open)(const char *, struct fuse_file_info *)
```

read lê os dados de um arquivo aberto

```
int(* fuse_operations::read)(const char *, char *, size_t, off_t, struct fuse_file_info *)
```

release libera um arquivo aberto, é chamada quando não serão mais feitas referências à ao arquivo que está aberto

```
int(* fuse_operations::release)(const char *, struct fuse_file_info *)
```

destroy limpa o sistema de arquivos, chamado quando sair do sistema de arquivos.

```
void(* fuse_operations::destroy)(void *)
```

Em nosso trabalho usamos a nomenclatura da função, utilizando seu próprio nome com o prefíxo “fat16”.

Ex: para a função *readdir* implementamos *fat16_readdir*.

Para anexar estas funções implementadas ao fuse, devermos então atribuí-las à estrutura fuse_operations como mostrado à seguir:

```
struct fuse_operations fat16_oper = {
  .init       = fat16_init,
  .getattr    = fat16_getattr,
  .opendir    = fat16_opendir,
  .readdir    = fat16_readdir,
  .releasedir = fat16_releasedir,
  .open       = fat16_open,
  .read       = fat16_read,
  .release    = fat16_release,
  .destroy    = fat16_destroy
};
```

5.2. Funcionamento

Para compilar e executar o programa, utiliza-se os seguintes comandos:

```
make
./mount_fat16 <diretório de montagem>
```

Ao executar estes comandos, a FUSE chama a função *init* primeiramente, inicializando o sistema de arquivo, logo depois, o *getattr*, obtedo os atributos do diretório raíz.

Ao efetuar uma listagem com o comando ls, será chamada a função *readdir* que percorrerá as entradas do diretório, listando cada entrada de diretório correspondente à arquivos e diretórios encontrados. Quanto ao readdir, ele executa diferentemente no diretório raíz e em outros diretórios, já que no diretório raíz, ele deve percorre-lo acessando apenas a região reservada, e em outros diretórios, que se encontram na região de dados, deverão fazer acesso a FAT para consultar se há mais informação a ser lida ou se o último cluster é o que está sendo lido.

Ao entrar em um novo subdiretório com o comando cd, primeiro deverá ser executado o *readdir* para ler as entradas de diretório do diretório atual, então busca-se o diretório a ser encontrado utilizando o path, e ao encontrar o diretório, executa-se *releasedir* no diretório atual e *opendir* no diretório encontrado.

Para efetuar um uma cópia com o comando cp de dentro da imagem para fora, deve-se abrir o arquivo identificado com a função *open*, ler o seu conteúdo com a função *read*, e assim os dados já estarão em buffer e serão transferidos para o destino, onde o

FUSE fará a comunicação com o kernel para que a escrita seja feita no outro sistema de arquivos. Após finalizar o *read*, o arquivo deve ser liberado com a função *release*.

No fechamento do sistema de arquivo, deverá ser chamada a função *destroy*.

6. REFERÊNCIAS

- libfuse: fuse_operations Struct Reference (http://libfuse.github.io/doxygen/structfuse__operations.html)
- Write a filesystem with FUSE (https://engineering.facile.it/blog/eng/write-filesystem-fuse/)
- Microsoft FAT Specification, Microsoft Corporation, August 30 2005
- Repósitório do github libfat com implementação da FAT (https://github.com/Hexxeh/libfat)