UNIVERSIDADE FEDERAL DE GOIÁS – UFG
INSTITUTO DE INFORMÁTICA – INF

Arthur de Oliveira Barbosa Lacerda
Murillo Rodrigues de Paula

Relatório de desenvolvimento do sistema de arquivos FAT16

Sistemas Operacionais 2
Prof. Dr. Bruno Oliveira Silvestre

Goiânia, 2017

SUMÁRIO

1. CONVENÇÕES DE CÓDIGO

- Todas as variáveis utilizadas foram nomeadas na língua inglesa e apresentam os seguintes formatos:
  - Ex: variableNameExample ou VariableNameExample
- Toda função foi nomeada na língua inglesa, utilizando apenas letras minusculas e tem o caracter “_” como separador.
  - Ex: function_example
- Todo comentário do código também está na língua inglesa.
- Tamanho 2 de tabulação.
  - 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 */
```

2. ESTRUTURAS E TIPOLOGIAS UTILIZADAS

Por questão de compatibilidade e uniformidade com a FAT16, todas os tipos são sem sinal (`unsigned`) por padrão. Não utilizamos uma estrutura FAT porque a função `fat_entry_by_cluster` descrita na seção 4.1.4 realiza o funcionamento de consulta à região da primeira FAT. Tentamos seguir todas as recomendações e especificações oficiais da FAT encontradas em [1] e [5].

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.

A estrutura `BPB_BS` descreve uma estrutura BPB da FAT16.
A estrutura `DIR_ENTRY` descreve uma estrutura de arquivo/diretório da FAT16.
A estrutura `VOLUME` descreve uma estrutura que contém dados essenciais da FAT16.

Segue abaixo as estuturas e definições utilizadas:

```
#define BYTES_PER_DIR 32
#define ATTR_DIRECTORY 0x10
#define ATTR_ARCHIVE 0x20

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* 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, a partir do SEEK_SET (ponto inicial do arquivo imagem 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. ESTUDOS E TESTES COM UMA IMAGEM FAT16

Primeiramente escrevemos o programa **run_fat16.c** a fim de entender o funcionamento da FAT16. Com isso, fomos capazes de caminhar pelos seus diretórios e ver os atributos dos seus arquivos e diretórios. Nesta seção descreveremos como conseguimos realizar testes com o sistema de arquivos FAT16 apenas manipulando um arquivo-imagem FAT16 independente. Detalhes de como o implementamos com o FUSE (**mount_fat16.c)** se encontram na seção 5. Todos os testes foram realizados em um ambiente com o sistema operacional GNU/Linux Ubuntu 17.04 de 64 bits.

O programa run_fat16.c recebe como parâmetro a imagem da FAT16 e o caminho a ser percorrido. É apresentado na saída se o caminho foi ou não encontrado com sucesso na imagem. Mais detalhes específicos de como esse programa funciona estão presentes na seção 4.2.

4.1. Funções implementadas

4.1.1. `path_treatment`

retorno:

`char** pathFormatted`: Vetor de strings com cada string referente a 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 a variável `pathFormatted` com as devidas alterações, sendo que ela aceita apenas os caracteres legais pela especificação da FAT. É trocado caracteres em caixa baixa por caixa alta, os 8 primeiros espaços das string são referentes ao nome do arquivo e os três últimos referentes a extensão. Qualquer espaço inutilizado é preenchido com o caracter de espaço em branco “ “. Também é tratado o caso em que os arquivos possam ser “.” ou “..” (somente no programa teste run_fat16.c).

4.1.2. `pre_init_fat16`

retorno:

`VOLUME* Vol`: Estrutura com dados essenciais da FAT16

parâmetros:

1. void

descrição:

Abre o arquivo da imagem FAT16 no modo “rb” e grava seu descritor (Vol→fd) para uso no decorrer do programa.

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:

0, se é achado um arquivo correspondente ao path ou 1 caso contrário

parâmetros:

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

`WORD`

parâmetros:

1. `VOLUME* Vol`: Estrutura com dados essenciais da FAT16.
2. `WORD ClusterN`: Cluster em que será usado como base para obter a entrada da FAT.

descrição:

Com essa função, não é feito necessário o uso de uma estrutura FAT para armazenar setores da região FAT, pois a partir de qualquer cluster número N podemos obter sua entrada correspondente na FAT. O procedimento é o seguinte:

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 (16 bits).

Calcula-se então o número do setor da FAT em que a leitura deve ser realizada, saltando a área reservada do diretório da raiz acrescida do número de setores do offset da FAT (`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 ao numero do setor calculado e retorna o valor encontrado na entrada da FAT.

4.1.5. `find_subdir`

retorno:

`void`

parâmetros:

1. `VOLUME Vol`: Estrutura com dados essenciais da FAT16.
2. `DIR_ENTRY Dir` Variável que armazenará entradas de diretório situadas no subdiretório.
3. `char** path`
4. `int pathSize`: Vetor de arquivos do caminho à ser percorrido.
5. `int pathDepth`: Profundidade (ou índice) do caminho.
6. `int rootDepth`: Profundidade atual do caminho em relação ao diretório raiz (ausente no programa mount_fat16.c, pois foi usada somente em run_fat16.c para entender o funcionamento da FAT).

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 as 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. `run_fat16.c`: funcionamento

Para compilar e executar o programa, deve-se utilizar a sequência de comandos a seguir:

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

Na função `main` do programa run_fat16.c 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*, a fim de armazenar o BPB e determinar os setores de início do diretório raiz e da área de dados. 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. A recursão voltará a *find_root* caso a profundidade relativa ao diretório raiz seja zero. Ao encontrar o arquivo final no 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 existe tal arquivo.

Abordando em um nível estrutural, primeiramente é lido o primeiro setor da FAT e então salta-se para o primeiro setor do diretório raiz. Depois busca-se entrada por entrada no diretório raiz até que seja correspondente ao arquivo encontrado. Caso a entrada não seja encontrada no diretório raiz, 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 na região de dados e na FAT. No programa teste run_fat16.c, ocasionalmente a recursão poderá voltar à região do diretório raíz.

5. FUSE – IMPLEMENTAÇÃO COM FAT16

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)`

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)`

read lê os dados de um arquivo aberto
`int(* fuse_operations::read)(const char *, char *, size_t, off_t, 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”.

Todas as implementações foram realizadas com base na documentação das estruturas e funções do FUSE [2], além de códigos exemplos disponibilizados no repositório oficial do FUSE [3] e de manuais de implementação não-oficiais [4], [6]. Outras funções não essenciais não foram implementadas, pois não eram necessárias para atingir os objetivos do trabalho.

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

```
struct fuse_operations fat16_oper = {
  .init       = fat16_init,
  .destroy    = fat16_destroy,
  .getattr    = fat16_getattr,
  .readdir    = fat16_readdir,
  .read       = fat16_read
};
```

5.2. `mount_fat16.c`

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

```
1. make
2. ./mount_fat16 <diretório de montagem> -s
```

A opção `-s` é necessária para desabilitar a operação multi-threaded, visto que o programa não foi projetado para dar suporte a paralelismo. Se não for fornecida a opção `-s`, o programa irá mal funcionar na operação de cópia de um arquivo de dentro do sistema de arquivos para fora com a função `fat16_read` porque a função `fuse_main` fará várias chamadas à chamada de sistema `fork()` em modo *daemon.* Mais detalhes dessa função na seção 5.4.5.

Ao executar esses 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 ou percurso com os comandos `ls, cd`, ou mesmo por uma GUI, será chamada a função `readdir` que percorrerá as entradas do diretório, listando cada entrada de diretório correspondente a arquivos e diretórios encontrados. Quanto ao readdir, ele executa o percurso diferentemente no diretório raíz do que em outros diretórios, já que no diretório raíz ele deve percorrê-lo acessando apenas a região reservada da raiz (que tem tamanho fixo de entradas de diretórios especificado pelo campo `BPB_RootEntCnt`). Em outros diretórios, que se encontram na região de dados, deverá ser feito acesso a FAT para consultar se há mais informação a ser lida ou se o último cluster é o que está sendo lido.

Para efetuar um uma cópia (com o comando `cp`, por exemplo) de dentro da imagem para fora, é usada a função `fat16_read`, que primeiro deve encontrar a entrada de diretório com `find_root` (que recursivamente poderá chamar `find_subdir`) e depois lê bytes do caminho dado para o buffer (mais detalhes na seção 5.4.5). Com os bytes já armazenados no buffer, o FUSE pode então comunicar com o kernel do Linux para copiar esses bytes de dentro da imagem para fora dela.

No fechamento do sistema de arquivo, deverá ser chamada a função `destroy` que irá liberar os dados passados como parâmetro.

Após a utilização do sistema de arquivos, basta desmontá-lo através do comando:

3. `fusermount -u <diretório de montagem>`

5.3. Reutilização das funções de `run_fat16.c`

As funções *find_root, find_subdir* e *path_treatment* foram reutilizadas na implementação do FUSE, porém com algumas alterações funcionais:

path_treatment não precisa mais verificar se a entrada é válida, apenas realiza o tratamento para o formato padrão.

find_root e *find_subdir* não imprimem mais o diretório, mas o armazenam, passando um ponteiro DIR_ENTRY como parâmetro, e retornam 0 em caso de sucesso e 1 em caso de fracasso.

fat_entry_by_cluser foi reutilizada sem alterações funcionais.

5.4. Funções implementadas

5.4.1. `path_decode`

retorno:

`char *pathDecoded`: Nome do arquivo no formato comum, com nome seguindo de ponto e extensão em letras minúsculas.

parâmetros:

1. `path`: Nome do arquivo no formato FAT16

descrição:

A função passa o formato de nomenclatura da FAT para letras minúsculas e nome e extensão separados por ponto, respeitando os caracteres especiais legalizados de acordo com a especificação da FAT.

Ex: “`ARQ TXT`” -> “`arq.txt`”

5.4.2. `fat16_init`

retorno:

`void* context->private_data`: ponteiro para os dados do usuário FUSE retornados pela função init.

parâmetros:

1. `struct fuse_conn_info *conn`: fornece informação sobre quais características tem suporte pelo FUSE

descrição:

realiza uma preparação inicial de única vez e recebe o contexto.

5.4.3. `fat16_getattr`*:* atributos de arquivos e diretórios

retorno:

retorna o valor 0 no fim da função.

parâmetros:

1. `const char *path`: caminho do arquivo que os atributos devem ser obtidos.
2. `struct stat *stbuf`: estrutura que armazena os atributos do arquivo.

descrição:

A função ao receber o caminho, verifica primeiramente se o path é "/", ou seja, se o arquivo à obter os atributos é o diretório raiz, se sim, atribui-se os devidos valores, se não, realiza o `path_treatment`, e então chama `find_root` que retornará a entrada de diretório correspondente ao path, e então os atributos são obtidos à partir da entrada de diretório.

5.4.4. `fat16_readdir`*:* listando diretórios

retorno:

retorna o valor 0 no fim da função

parâmetros:

1. `const char *path`: caminho do diretório a ser lido.
2. `void *buffer`: responsável para passar as informações para a FUSE.
3. `fuse_fill_dir_t filler`: preenche o buffer com os nomes dos arquivos do diretório.
4. `off_t offset`: não utilizado na nossa implementação
5. `struct fuse_file_info *fi`: não utilizado na nossa implementação

descrição:

Se o path for "/", ele percorrerá o diretório raiz e passará para o filler todo nome de diretório ou arquivo que forem encontrados em entradas do diretório raiz.

Se o path não for "/", o path passará pelo path_treatment e percorrerá com *find_root* o path dado de entrada e assim obtém a entrada de diretório destino. Com a entrada de diretório, encontra-se o primeiro cluster e inicia a leitura das entradas chamando a função `filler` para cada uma encontrada (em cada nome, usa-se `path_decode` para decodificar o nome da FAT*).* Se existir mais de um cluster, a cada finalização de leitura de cluster a entrada de FAT é acessada para obter o novo cluster e continuar a leitura, até que a entrada da FAT indique que não há mais clusters a serem

lidos para aquele determinado diretório. Note que não é necessário fazer essa verificação de fim de cluster porque um caminho fornecido pelo FUSE sempre estará presente em algum diretório, uma vez que foi passado ao FUSE pela função `filler`, mas optamos por incluí-la para ficar imune a quaisquer erros eventuais gerados pelo FUSE que não compremeta o funcionamento dessa função).

5.4.5. `fat16_read`: copiando arquivos da FAT16 para fora

retorno:

int size: Tamanho do arquivo a ser lido

parâmetros:

1. `const char *path`: caminho do arquivo à ser lido.
2. `char *buffer`: buffer que guardará os dados a serem gravados.
3. `size_t size`: tamanho do buffer em bytes.
4. `off_t offset`: offset de leitura.
5. `struct fuse_file_info *fi`: não utilizado na nossa implementação

descrição:

Primeiramente, devemos desabilitar a opção do uso de múltiplas threads do FUSE com a opção `-s` na execução do programa. Isso porque a biblioteca FUSE cria múltiplos processos filhos com a função `fork()` através da função `fuse_main,` com o propósito de executar a função `fat16_read` de modo paralelo. Na nossa implementação, toda leitura com a função `fat16_read` é feita de forma sequencial com base nas chamadas individuais a essa função pelo FUSE. Caso a opção `-s` não seja usada, com os testes realizados no nosso ambiente, o comportamento esperado do programa é normal exceto quando é feita uma leitura de um arquivo maior que 140KB. Sem essa opção, se uma leitura a um arquivo maior que 140KB é feita, o comportamento do programa é inesperado quando a função `fat16_read` é chamada mas o funcionamento será normal para todas as outras operações.

Com o `path`, usamos `find_root` para encontrar o arquivo e acessamos seu primeiro setor do cluster. A cada iteração para leitura dos bytes do arquivo, `sector_read` é chamado (`size + offset - BYTES_PER_SECTOR`) vezes com alternância de clusters do arquivo, até que não tenha mais conteúdo a ser lido.

Com o `offset`, nas chamadas subsequentes dessa função pelo FUSE, o arquivo continuará a ser lido do seu ponto inicial até `size + offset` usando o mesmo procedimento de leitura descrito anteriormente. No final da leitura, retornamos o número de bytes requisitado ou 0 se `offset` for igual ou maior que o número de bytes do arquivo. Nota-se que essa solução não é paralela mas sim sequencial e portanto não faz o uso das várias threads disparadas por `fuse_main`.

Assim, conseguimos atingir o objetivo final do trabalho de copiar um arquivo de dentro da imagem FAT16 para fora dela usando a biblioteca FUSE.

6. REFERÊNCIAS

- [1] libfuse: fuse_operations Struct Reference (http://libfuse.github.io/doxygen/structfuse__operations.html)
- [2] How FAT Works (https://technet.microsoft.com/en-us/library/cc776720(v=ws.10).aspx)
- [3] (https://github.com/libfuse/libfuse)
- [4] Write a filesystem with FUSE (https://engineering.facile.it/blog/eng/write-filesystem-fuse/)
- [5] Microsoft FAT Specification, Microsoft Corporation, August 30 2005
- [6] CS135 FUSE Documentation (https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201001/homework/fuse/fuse_doc.html)