Universidade Federal de Minas Gerais
DCC605: Sistemas Operacionais
Trabalho Prático

Cronograma e execução

Execução: individual
Valor: **50 pontos**

Sistema de Arquivos: DCC605FS

Neste trabalho você irá implementar as primitivas de um sistema de arquivos, o DCC605FS. O DCC605FS é armazenado dentro de um arquivo no sistema operacional. Da mesma forma que você pode abrir um arquivo compactado (zip) e navegar pelo seu conteúdo, um sistema operacional poderá montar um arquivo contendo um sistema de arquivos DCC605FS e navegar pelo seu conteúdo.

Interface do sistema de arquivos

Neste trabalho você irá implementar parte das funções do DCC605FS; em particular, as funções para formatar um arquivo em disco com um novo DCC605FS e para retornar um bloco livre no sistema de arquivos. De forma mais detalhada, as funções que você deve implementar são:

```
struct superblock * fs_format(const char *fname, uint64_t blocksize);
```

Constrói um novo sistema de arquivos no arquivo `fname`. O arquivo `fname` deve existir no sistema de arquivos do sistema operacional. O sistema de arquivos DCC605FS criado em `fname` deve usar blocos de tamanho `blocksize`; o número de blocos no sistema de arquivos deve ser calculado automaticamente baseado no tamanho do arquivo `fname`. O sistema de arquivos deve ser inicializado com um diretório raiz vazio; o nome do diretório raiz deve ser `"/"` como em sistemas UNIX.

Esta função retorna NULL em caso de erro e guarda em errno o código de erro apropriado. Se o tamanho do bloco for menor que `MIN_BLOCK_SIZE`, a função falha e atribui `EINVAL` a `errno`. Se existir espaço em `fname` insuficiente para armazenar `MIN_BLOCK_COUNT` blocos, a função falha e atribui `ENOSPC` a `errno`.

```
struct superblock * fs_open(const char *fname);
```

Abre o sistema de arquivos em `fname` e retorna seu superbloco. Retorna NULL em caso de erro e carrega o código de erro em `errno`. Caso o superbloco em `fname` não contenha o marcador de sistemas de arquivo DCC605FS (`0xdcc605f5`), a função falha e atribui `EBADF` a `errno`.

Note que suas funções `fs_format` e `fs_open` não devem permitir que o mesmo sistema de arquivo seja aberto duas vezes, para evitar corrupção dos arquivos. No caso do sistema de arquivos já estar aberto, estas funções devem falhar e atribuir `EBUSY` a `errno`.

```
int fs_close(struct superblock *sb);
```

Fecha o sistema de arquivos apontado por `sb`. Libera toda a memória e recursos alocados à estrutura `sb`. Retorna zero se não houver erro. Em caso de erro, um valor negativo é retornado e `errno` indica o erro. Se `sb` não tiver o marcador `0xdcc605f5`, atribui `EBADF` a `errno`.

```
uint64_t fs_get_block(struct superblock *sb);
```

Pega um ponteiro para um bloco livre no sistema de arquivos `sb`. O bloco retornado é retirado da lista de blocos livres do sistema de arquivos. Retorna zero caso não existam mais blocos livres; retorna `(uint64_t)-1` e atribui `errno` caso um erro ocorra.

```
int fs_put_block(struct superblock *sb, uint64_t block);
```

Retorna block para a lista de blocos livres do sistema de arquivo `sb`. Retorna zero em caso de sucesso e um valor negativo em caso de erro. Código de erro, se ocorrer, é salvo em `errno`.

```
int fs_write_file(struct superblock *sb, const char *fname, char *buf,
                  size_t cnt);
```

Escreve `cnt` bytes de `buf` no sistema de arquivos apontado por `sb`. Os dados serão escritos num arquivo chamado `fname`. O parâmetro `fname` deve conter um caminho absoluto. Retorna zero em caso de sucesso e um valor negativo em caso de erro; em caso de erro, este será salvo em `errno` (p.ex., espaço em disco insuficiente). Se o arquivo já existir, ele deverá ser sobrescrito por completo com os novos dados.

```
ssize_t fs_read_file(struct superblock *sb, const char *fname, char *buf,
                     size_t bufsz);
```

Lê os primeiros `bufsz` bytes do arquivo fname e coloca no vetor apontado por `buf`. Retorna a quantidade de bytes lidos em caso de sucesso (pode ser menos que `bufsz` se o arquivo for menor que `bufsz`) e um valor negativo em caso de erro. Em caso de erro a variável `errno` deve ser utilizada para indicar qual erro aconteceu.

```
int fs_unlink(struct superblock *sb, const char *fname);
```

Remove o arquivo chamado `fname` do sistema de arquivos apontado por `sb` (os blocos associados ao arquivo devem ser liberados). Retorna zero em caso de sucesso e um valor negativo em caso de erro; em caso de erro, este será salvo em `errno` de acordo com a função `unlink` em `unistd.h` (p.ex., arquivo não encontrado).

```
int fs_mkdir(struct superblock *sb, const char *dpath);
```

Cria um diretório no caminho `dpath`. O caminho `dpath` deve ser absoluto (começar com uma barra). Retorna zero em caso de sucesso e um valor negativo em caso de erro; em caso de erro, este será salvo em `errno` de acordo com a função `mkdir` em `unistd.h` (p.ex., diretório já existente, espaço em disco insuficiente). O caminho `dpath` não deve conter espaços. A função `fs_mkdir` *não* precisa criar diretórios recursivamente; ela cria apenas um diretório por vez. Para criar o diretório `/x/y` é preciso antes criar o diretório `/x`; se o diretório `/x` não existir, retorno `ENOENT`.

```
int fs_rmdir(struct superblock *sb, const char *dname);
```

Remove o diretório no caminho `dpath`. O caminho `dpath` deve ser absoluto (começar com uma barra). Retorna zero em caso de sucesso e um valor negativo em caso de erro; em caso de erro, este será salvo em `errno` de acordo com a função `rmdir` em `unistd.h` (p.ex., `ENOTEMPTY` se o diretório não estiver vazio). O caminho `dpath` não deve conter espaços.

```
char * fs_list_dir(struct superblock *sb, const char *dname);
```

Retorna um string com o nome de todos os elementos (arquivos e diretórios) no diretório `dname`, os elementos devem estar separados por espaço. Os diretórios devem estar indicados com uma barra ao final do nome. A ordem dos arquivos no string retornado não é relevante. Por exemplo, se um diretório contém três elementos--um diretório `d1` e dois arquivos `f1` e `f2`--esta função deve retornar:

“`d1/ f1 f2`”

Estruturas do sistema de arquivos

Para facilitar a codificação do DCC605FS, várias estruturas de dados já foram definidas. Você deve manter estas estruturas de dados como definidas para permitir interoperação de diferentes implementações do DCC605FS. Para facilitar, você pode assumir que o DCC605FS só será utilizado em computadores *little-endian* como o Intel x86 e x64.

struct superblock

```
struct superblock {
	uint64_t magic; /* 0xdcc605f5 */
	uint64_t blks; /* number of blocks in the filesystem */
	uint64_t blksz; /* block size (bytes) */
	uint64_t freeblks; /* number of free blocks in the filesystem */
	uint64_t freelist; /* pointer to free block list */
	uint64_t root; /* pointer to root directory's inode */
	int fd; /* file descriptor for the filesystem image */
};
```

O superbloco é sempre armazenado no primeiro bloco do arquivo onde o DCC605FS foi criado. Os primeiros 64bits (campo `magic`) devem conter o valor `0xdcc605f5`. O campo `root` deve conter um ponteiro para o bloco que armazena o diretório raiz do sistema de arquivo. O inteiro `fd` é utilizado para armazenar o descritor do arquivo utilizado para montar o sistema de arquivo (via `fs_open` ou `fs_format` (durante a formatação)).

struct inode

```
struct inode {
	uint64_t mode;
	uint64_t parent;
	/* if =mode does not contain IMCHILD, then =parent points to the
	 * directory that contains this inode.  if =mode contains IMCHILD,
	 * then =parent points to the first inode (i.e., the inode without
	 * IMCHILD) for the entity represented by this inode. */
	uint64_t meta;
	/* if =mode does not contain IMCHILD, then meta points to this
inode's
	 * metadata (struct iinfo).  if =mode contains IMCHILD, then meta
	 * points to the previous inode for this inode's entity. */
	uint64_t next;
	/* if this file's date block do not fit in this inode, =next points
to
	 * the next inode for this entity; otherwise =next should be zero. */
	uint64_t links[];
```

```
	/* if =mode contains IMDIR, then entries in =links point to inode's
	 * for each entity in the directory.  otherwise, if =mode contains
	 * IMREG, then entries in =links point to this file's data blocks. */
};
```

A mesma estrutura inode acima é utilizada para armazenar diretórios e arquivos. Em diretórios, os `links` apontam para inodes das entidades (arquivos e diretórios) contidos no diretório. Em arquivos, os `links` são *ordenados* e apontam para os blocos contendo os dados do arquivo. O campo `mode` diferencia entre os dois casos com as máscaras:

```
#define IMREG 1   /* regular inode (normal file) */
#define IMDIR 2   /* directory inode */
```

Diretórios com muitas entidades e arquivos muito grandes podem requerer quantidade de entradas no vetor `links` maior que disponível em um único bloco. Neste caso, o DCC605FS cria uma lista encadeada de `inodes` para representar a entidade. O campo `next` aponta para o próximo `inode` na representação da entidade (ou tem o valor zero se o `inode` for o último). Todos os `inodes` de uma entidade que não são o primeiro possuem a constante `IMCHILD` em seu campo `mode`:

```
#define IMCHILD 4 /* child inode */
```

A semântica dos campos `parent` e `meta` de um `inode` varia como dependendo se o `inode` for `IMCHILD` ou não. Para `inodes` `IMCHILD` o campo `parent` aponta para o primeiro `inode` da entidade (o `inode` que não é `IMCHILD`); o campo `meta` aponta para o `inode` anterior na representação da entidade. Para `inodes` que não são `IMCHILD` o campo `parent` aponta para o `inode` principal (sem `IMCHILD`) do diretório que contém a entidade; o campo `meta` aponta para o `inode` que contém os metadados do arquivo (`struct nodeinfo`).

struct nodeinfo

```
struct nodeinfo {
	uint64_t size;
	/* for files (mode IMREG), =size should contain the size of the file
in
	 * bytes.  for directories (mode IMDIR), =size should contain the
	 * number of files in the directory. */
	uint64_t reserved[7];
	/* reserving some space to implement security and ownership in the
	 * future. */
	char name[];
	/* remainder of block used to store this entity's name. */
};
```

A estrutura `struct nodeinfo` armazena metadados de uma entidade. Por enquanto só armazenamos o tamanho e o nome da entidade. Para arquivos (`IMREG`), size deve conter o tamanho do arquivo em bytes. Para diretórios (`IMDIR`), size deve conter o número de entidades no diretório. A cadeia de caracteres em `name` deve terminar com o caractere nulo (‘`\0`’).

struct freepage

```
struct freepage {
      uint64_t next;
      /* link to next freepage; or zero if this is the last freepage */
      uint64_t count;
      uint64_t links[];
      /* remainder of block used to store links to free blocks.  =count
       * counts the number of elements in links, stored from links[0] to
       * links[counts-1]. */
};
```

A `struct freepage` serve para armazenar os blocos de dados livres. Como `inodes`, as páginas contendo os blocos de dados livres são encadeadas através dos campos `next`. O vetor `links` contém ponteiros para blocos livres no sistema de arquivos. O campo `count` contém o número de blocos livres *nesta* página e serve para indexar o vetor `links`.

Importante. Os ponteiros para blocos no sistema de arquivo têm 64bits e contam o número de blocos no sistema de arquivo a partir de zero (não contam número de bytes).

Importante. A manutenção dos blocos livres não deve utilizar quantidade arbitrariamente grande de blocos no disco. Em particular, o teste padronizado confere se a quantidade de blocos utilizados num sistema de arquivos vazio é menor que seis; idealmente, a manutenção de blocos livres deve utilizar os blocos livres.

Desenvolvimento e Avaliação

Você pode baixar o arquivo fs.h contendo as definições acima. Sua avaliação será feita através de uma versão estendida de um teste padronizado que você pode utilizar para testar seu código.

Se você identificar falhas ou ambiguidades ou problemas nesta especificação, descrevê-las ao professor pode resultar na atribuição de pontos extras. Note que a estrutura inicial do sistema de arquivos e os algoritmos para controle do espaço livre foram deixados em aberto intencionalmente.

Relatório

Este trabalho será entregue em duas partes. Na primeira parte deve ser entregue apenas o arquivo fs.c implementando as funções especificadas. Na segunda parte deve ser entregue o arquivo [report.txt] incluso no repositório preenchido. Na segunda parte você deve entregar também uma documentação de 4 páginas (2 folhas) em fonte 10 pontos, descrevendo (*com figuras*) a organização inicial do seu sistema de arquivos e a organização das páginas de blocos livres (*com figuras também*). Sua documentação também deve discutir (e justificar) as escolhas realizadas.