SISTEMA DE FICHEIROS sofs14

Nota prévia

sofs14 é um sistema de ficheiros simples e limitado, baseado no sistema de ficheiros *ext2* do Linux, que foi concebido com propósitos meramente didácticos e é destinado a ser desenvolvido nas aulas práticas de Sistemas de Operação no ano lectivo de 2014/2015.

O suporte físico preferencial é um ficheiro regular do sistema de ficheiros da plataforma hardware que vai ser usada no seu desenvolvimento.

A infraestrutura FUSE

Em termos gerais, a introdução de um novo sistema de ficheiros no sistema de operação implica a realização de duas tarefas bem definidas. A primeira consiste na integração do código associado à implementação do tipo de dados *ficheiro* no núcleo, *kernel*, do sistema de operação e, a segunda, na sua instanciação sobre um ou mais dispositivos de memória de massa do sistema computacional.

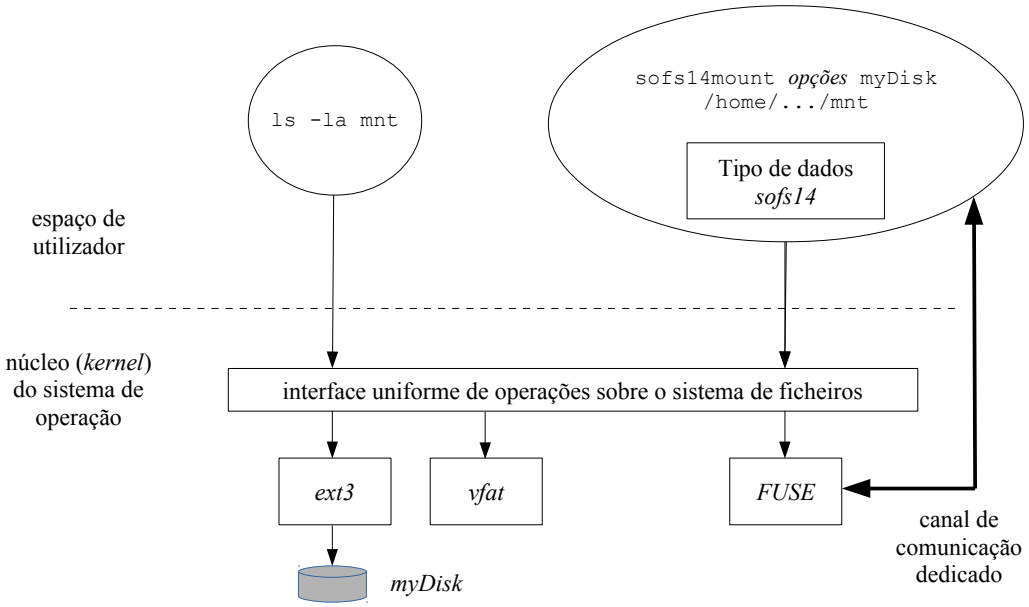
Em situações de *kernel* monolítico, a integração do código traduz-se na criação de um novo *kernel* através da compilação e *linkagem* dos diferentes ficheiros fonte descritivos do sistema de operação. Em situações de *kernel* modular, o módulo associado é compilado e *linkado* separadamente, sendo acoplado a um *kernel* pré-existente em *run time*. Qualquer que seja o caso, porém, trata-se de uma tarefa muito complexa e especializada que exige um conhecimento profundo das estruturas de dados internas e da funcionalidade apresentada pelo sistema de operação-alvo, estando habitualmente só estão ao alcance de programadores de sistemas experientes.

FUSE (*File system in User SpacE*) constitui uma solução alternativa muito engenhosa que visa a construção de sistemas de ficheiros no *espaço de utilizador*, como se se tratasse de meras aplicações, impedindo que eventuais inconsistências e erros que surjam na sua implementação, sejam transmitidos directamente ao *kernel* e conduzam à sua inoperacionalidade.

A infraestrutura oferecida é formada por duas partes principais:

- *módulo de interface com o sistema de ficheiros* – actua como o mediador de comunicações entre o *interface uniforme de operações sobre ficheiros* disponibilizado pelo *kernel* e a respectiva implementação em *espaço de utilizador*;
- *biblioteca de implementação* – fornece as estruturas de dados de comunicação e o protótipo das operações que têm que ser desenvolvidas para garantir a compatibilidade operacional do tipo de dados definido pelo utilizador com o modelo subjacente; fornece, além disso, um conjunto de funcionalidades destinadas a instanciar o tipo de dados sobre um dispositivo de memória de massa e à sua integração no sistema de operação.

O diagrama abaixo ilustra o tipo de organização que resulta quando o sistema de ficheiros *sofs14*, instanciado sobre o dispositivo *myDisk* que representa aqui um ficheiro do sistema de ficheiros *ext3* de Linux, é integrado no sistema de operação usando o FUSE.



É responsabilidade do programador de aplicações estabelecer a compatibilização da semântica de operações providenciada por FUSE com a implementação do tipo de dados *sistema de ficheiros* (*sofs14*, no caso vertente). Assim, a estrutura de dados **`struct`** `fuse_operations`, definida em */include/fuse/fuse.h*, é preenchida com a especificação das operações correspondentes

```
static struct fuse_operations op =
   { .getattr = sofs_getattr,
     .readlink = sofs_readlink,
     .getdir = sofs_getdir,
     .mknod = sofs_mknod,
     .mkdir = sofs_mkdir,
     .unlink = sofs_unlink,
     .rmdir = sofs_rmdir,
     .symlink = sofs_symlink,
     .rename = sofs_rename,
     .link = sofs_link,
     .chmod = sofs_chmod,
     .chown = sofs_chown,
     .truncate = sofs_truncate,
     .utime = sofs_utime,
     .open = sofs_open,
     .read = sofs_read,
     .write = sofs_write,
     .statfs = sofs_statfs,
     .flush = sofs_flush,
     .release = sofs_release,
     .fsync = sofs_fsync,
     .setxattr = sofs_setxattr,
     .getxattr = sofs_getxattr,
     .listxattr = sofs_listxattr,
     .removexattr = sofs_removexattr,
     .opendir = sofs_opendir,
     .readdir = sofs_readdir,
     .releasedir = sofs_releasedir,
     .fsyncdir = sofs_fsyncdir,
     .init = sofs_init,
     .destroy = sofs_destroy
   };
```

e a aplicação *sofs14mount*, ao ser executada, invoca como última instrução a função

```
int fuse_main (int, char **, const struct fuse_operations *, void *);
```

que procede à integração do sistema de ficheiros no sistema de operação e vai actuar no seguimento como gestor das operações que se realizam sobre ele.

A integração supõe duas fases

- estabelecimento de um canal de comunicação dedicado entre o módulo FUSE localizado no *kernel* e a aplicação *sofs14mount*;
- associação do directório */home/ ... /mnt* do sistema de ficheiros local ao descritor do canal de comunicação pela operação interna de montagem de um sistema de tipo FUSE sobre ele.

A partir daqui, qualquer operação efectuada sobre a sub-árvore de directórios que tem como base o directório */home/ ... /mnt*, é entendida como referente a um sistema de tipo FUSE pelo interface uniforme de acesso a ficheiros do sistema de operação e é dirigida, em consequência, ao módulo respectivo do *kernel*. Este módulo mantém um registo actualizado das diferentes operações de montagem que entretanto ocorreram e utiliza o canal de comunicação dedicado, que está associado à sub-árvore de directórios, para retransmitir a operação recebida à aplicação *sofs14mount*. Usando a tabela **`struct`** `fuse_operations op`, a aplicação converte a operação em uma ou mais operações do tipo de dados *sofs14*, executa-as sobre o dispositivo *myDisk* e envia os resultados de volta, pelo canal de comunicação dedicado, ao módulo FUSE do *kernel* que, por sua vez, os faz chegar à aplicação onde a operação teve origem.

Arquitectura do sistema de ficheiros sofs14

O ficheiro *sofs_const.h* especifica os valores de um conjunto de constantes básicas que caracterizam a abstracção do dispositivo de memória de massa como um *array* de blocos.

Em particular, o tamanho em bytes de cada bloco do dispositivo de memória de massa, *BLOCK_SIZE*, é colocado a 512 e impõe-se que as operações de reserva e de libertação de blocos para armazenamento de dados sejam realizadas sobre grupos de 4 blocos contíguos, aquilo que se designa de *cluster*.

Directórios

Para que a informação seja mais rapidamente acessível e se possam isolar regiões de armazenamento dentro da memória de massa, os sistemas de ficheiros são normalmente concebidos como uma hierarquia, ou *árvore*, de directórios.

Um *directório* constitui neste sentido um tipo particular de ficheiro cuja função é espelhar a descrição da estrutura hierárquica subjacente, contendo referências para os ficheiros que são visíveis a um dado nível da hierarquia interna.

O seu conteúdo informativo pode ser entendido como uma tabela de referências onde cada entrada associa o *nome* do ficheiro com o seu *identificador interno*.

O tipo de dados `SODirEntry` está na base desta concepção.

```
typedef struct soDirEntry
  {
    unsigned char name[MAX_NAME+1];  /* the name of a file (whether
                  a regular file, a directory or a symbolic link):
                                  it must be a NUL-terminated string */
    uint32_t nInode;                  /* the associated inode number */
  } SODirEntry;
```

Deve notar-se, porém, que esta tabela não tem um tamanho fixo.

Inicialmente, quando o directório é criado, a tabela tem *DPC* entradas; depois, à medida que o conteúdo informativo vai crescendo, o tamanho da tabela é incrementado, introduzindo-se de cada vez *DPC* novas entradas.

Um outro aspecto a ter em conta é que, para que seja possível navegar na árvore de directórios, um directório vazio não significa que todas as suas entradas estejam *livres*. As duas primeiras entradas estão sempre *ocupadas*: a primeira, a que é atribuído o nome "`.`", constitui uma auto-referência; a segunda, de nome "`..`", referencia o directório imediatamente acima na hierarquia. Uma entrada é considerada *livre* se pelo menos o primeiro caracter do campo `name` for o caracter '`\0`'.

Finalmente, a implementação de um mecanismo de recuperação de ficheiros anteriormente apagados, *undelete*, exige que as entradas livres possam estar em dois estados distintos: estado dito *limpo*, quando todos os caracteres do campo `name` são '`\0`' e o campo `nInode` tem o valor `NULL_INODE`, e estado dito *sujo*, quando só o primeiro caracter do campo `name` é igual a '`\0`'.

Esta convenção permite que, se um ficheiro for apagado, o primeiro e o último caracteres do campo `name` são trocados, o que transforma a entrada de *ocupada* em *livre*, no estado *sujo*, e possibilita, se necessário, a sua recuperação posterior. Por outro lado, se uma nova referência a um ficheiro for criada no directório, deve ser sempre feita sobre uma entrada *livre* no estado *limpo*.

Nós-i

Um *nó-i*, ou *nó identificador*, é a estrutura de dados onde estão alojados os restantes atributos de um ficheiro. Os nós-i estão organizados numa tabela interna de tamanho fixo. É importante notar que, como cada ficheiro do sistema de ficheiros tem os seus atributos armazenados num nó-i único, a sua localização na tabela (*array*) constitui o identificador interno do ficheiro e o número máximo de ficheiros distintos residentes no sistema de ficheiros é fixo e igual ao tamanho dessa tabela.

O nó-i é descrito pelo tipo de dados `SOInode`.

```
typedef struct soInode
  {
    uint16_t mode; /* it stores the file type (either a regular file,
                      a directory or a symbolic link) and its access
                     permissions:
                       bits 2-0 rwx permissions for other
                       bits 5-3 rwx permissions for group
                       bits 8-6 rwx permissions for owner
                       bit 9 is set if it represents a symbolic link
                       bit 10 is set if it represents a regular file
                       bit 11 is set if it represents a directory
                       bit 12 is set if it is free
                       the other bits are presently reserved       */
    uint16_t refCount;  /* number of hard links (directory entries)
                                        associated to the inode */
    uint32_t owner;                    /* user ID of the file owner */
    uint32_t group;                   /* group ID of the file owner */
    *         byte has been written */
    uint32_t size;   /* the farthest position from the beginning of
     the file information content + 1 where a byte has been written */
    uint32_t cluCount;    /* total number of data clusters attached
           to the file (this means both the data clusters that hold
            the file information content and the ones that hold the
                auxiliary data structures for indirect referencing) */
    union inodeFirst vD1;   /* context dependent variable of type 1 */
    union inodeSecond vD2;  /* context dependent variable of type 2 */
    uint32_t d[N_DIRECT]; /* direct references to the data clusters
                             that comprise the file information content */
    uint32_t i1;    /* reference to the data cluster that holds the
               group of direct references to the data clusters that
                            next comprise the file information content */
    uint32_t i2;     /* reference to the data cluster that holds an
                        array of indirect references holding in its
                groups of direct references to the data clusters that
                turn successive comprise the file information content */
} SOInode;
```

O campo `mode` serve simultaneamente para especificar o estado do nó-i, *ocupado* ou *livre*, o *tipo* do ficheiro descrito, *directório*, *ficheiro regular* ou *atalho*, e o *controlo de acesso* ao ficheiro, indicando, '`r`', quem pode ler o seu conteúdo (listar, se for um directório), '`w`', alterar o seu conteúdo (criar ou apagar ficheiros, se for um directório), ou, '`x`', realizar sobre ele operações de execução (aceder aos ficheiros referenciados, se for um directório). Os utilizadores estão divididos em três classes: o utilizador a que pertence o ficheiro, *owner*, os utilizadores incluídos no mesmo grupo do *owner*, *group*, e os utilizadores restantes, *other*.

Tal como para as entradas de directório, o mecanismo de recuperação de ficheiros anteriormente apagados impõe que os nós-i livres estejam em dois estados distintos: estado dito *limpo*, quando o nó-i não foi usado antes para descrever um ficheiro entretanto apagado, só o bit 12 está *set*, e estado dito *sujo*, em caso contrário, o bit 12 e um dos bits 9, 10 ou 11 estão *set*. Em princípio, quando um novo ficheiro é criado no sistema de ficheiros, deve ser usado um nó-i no estado *limpo* para conter a sua descrição; porém, se todos os nós-i livres estiverem *sujos*, aquele que for usado tem que ser colocado em primeiro lugar no estado *limpo*, o que significa que o ficheiro apagado, que era descrito por ele, não poderá mais vir a ser recuperado.

Torna-se muitas vezes conveniente manter referências para um dado ficheiro em diferentes regiões da árvore de directórios. Uma forma de se fazer isso, de uma forma indirecta, é criando um *atalho* que descreva a localização do ficheiro em causa na árvore de directórios, usando um *caminho absoluto* (iniciado na raiz do sistema de ficheiros) ou *relativo* (iniciado no directório onde é incluída a descrição). Esta solução tem, no entanto, um problema: quando o ficheiro referenciado é apagado, é deslocado de um directório para outro, ou o seu nome é modificado, os atalhos associados passam a referenciar coisa nenhuma e não é fácil, nem eficiente, estabelecer um procedimento que resolva a questão.

Um meio alternativo mais robusto e eficaz consiste no estabelecimento daquilo que se designa de *ligação primitiva* (ou *hard link*): a entrada de directório que é criada, em vez de referenciar o identificador interno de um ficheiro de texto, o *atalho*, que contém a descrição da localização do ficheiro em causa, passa a referenciar directamente o identificador interno do próprio ficheiro. Nestas condições, a alteração do nome ou a deslocalização de uma referência passam a ser problemas puramente locais da ligação primitiva envolvida na operação e não afectam as restantes ligações eventualmente existentes. Do mesmo modo, mantendo a contagem do número de ligações primitivas efectuadas, campo `refCount`, o apagamento de um ficheiro passa a ser uma operação que se desenvolve prioritariamente sobre a entrada de directório – o ficheiro só é realmente apagado quando o número de ligações primitivas se torna zero.

Os campos `owner` e `group` definem a relação de pertença e são usados, conjuntamente com os bits de *controlo de acesso* do campo `mode`, para estabelecer o tipo de operações que um utilizador genérico pode realizar sobre o ficheiro.

Ao definir-se uma estrutura de dados, acontece frequentemente que existem campos que só têm significado quando as variáveis desse tipo se encontram num estado bem definido. Nestas circunstâncias, é comum definir-se campos especiais, cujo significado depende do contexto da variável e em que, por questões de poupança de espaço, as diferentes interpretações ocupam o mesmo espaço de armazenamento. Este mecanismo foi usado aqui na especificação dos campos `vD1` e `vD2` e é descrito pelos tipos de dados **`union`** `inodeFirst` e **`union`** `inodeSecond`, respectivamente.

```
union inodeFirst
{
  uint32_t aTime;   /* if inode is in use, time of last file access */
  uint32_t next;   /* if inode is free, reference to the next inode
                          in the double-linked list of free inodes */
};

union inodeSecond
{
  uint32_t mTime;          /* if inode is in use, time of last file
                                                        modification */
  uint32_t prev;     /* if inode is free, reference to the previous
                      inode in the double-linked list of free inodes */
};
```

Os campos `vD1.aTime` e `vD2.mTime` constituem a monitorização de acesso ao ficheiro. Note-se que eles só têm este significado em nós-i *ocupados*. Em nó-i *livres*, devem ser interpretados, respectivamente, como `vD1.next` e `vD2.prev`, ponteiros que vão permitir implementar a *lista de nós-i livres* como uma lista biligada.

O acesso eficiente ao conteúdo informativo de um ficheiro, como um *continuum* de dados, supõe que se possa constituir uma lista ordenada de referências aos *clusters* da unidade de memória de massa onde a informação está armazenada. Tudo se passa como se o

*continuum*Arquitectura do sistema de ficheiros SOFS14 - 5

fosse dividido em pedaços de tamanho igual à capacidade de armazenamento de um *cluster* e a localização de cada *cluster* fosse armazenada no elemento de um *array d*, designado de *array de referências directas*, cujo índice representa o seu número de ordem.

Com efeito, sejam *p* a posição actual de um dado byte de informação no *continuum* de dados e *d*[*l*] e *off* as variáveis que permitem localizar o mesmo byte no *cluster* correspondente da memória de massa. As relações entre estas variáveis são expressas por

$$p = BSLPC \cdot l + off$$

$$l = p \text{ div } BSLPC$$

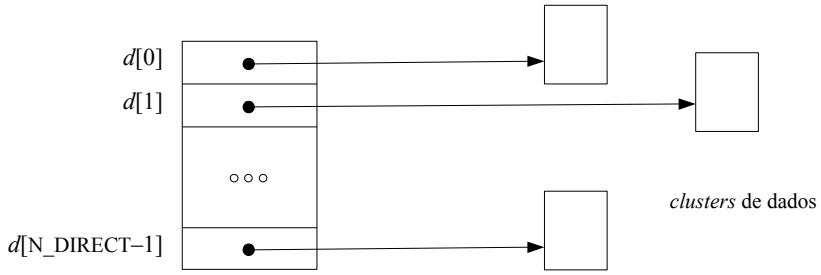
$$off = p \text{ mod } BSLPC$$

em que as operações *div* e *mod* representam, respectivamente, o *quociente* e o *resto da divisão inteira* dos operandos e *BSLPC* a capacidade de armazenamento de um *cluster*.

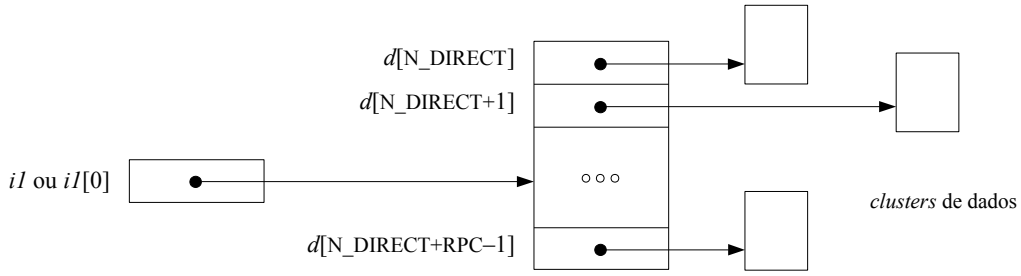
Assim sendo, o nó-i descritivo de um ficheiro terá que incluir de algum modo o *array d*. Contudo, se se pretender que o sistema de ficheiros possa conter ficheiros muito grandes, o tamanho desse *array* pode assumir um tamanho incomportável. De facto, no caso presente em que *BSLPC* é ligeiramente inferior a 2Kbytes, a possibilidade de se ter um ficheiro com um conteúdo informativo de 2Gbytes implicaria que o número de elementos do *array d* fosse da ordem de grandeza de um milhão!

O que se faz habitualmente nestas condições é estabelecer um compromisso entre o espaço necessário e a eficiência no acesso, prevendo mecanismos de referenciação indirecta de níveis progressivamente mais elevados.

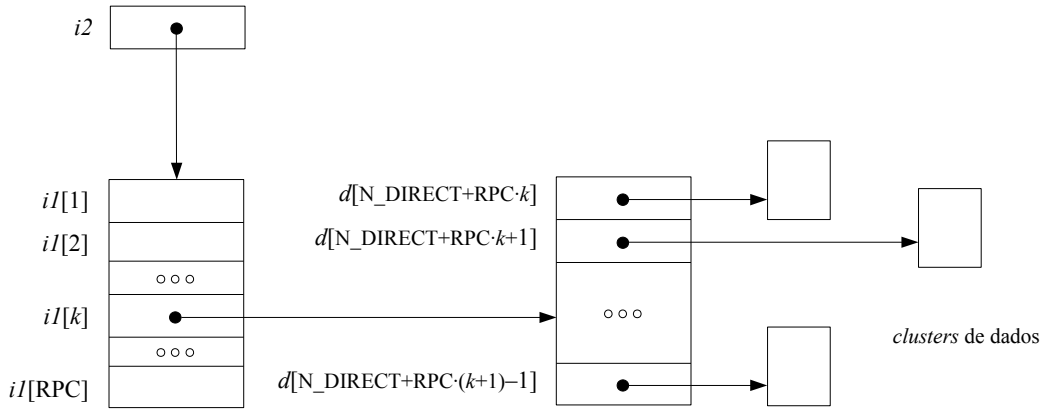
Referenciação directa



Referenciação simplesmente indirecta



Referenciação duplamente indirecta



No caso presente, em que foram implementados mecanismos de referenciação directa, simplesmente indirecta e duplamente indirecta através do *array* `d`[N_DIRECT] e das variáveis `i1` e `i2`, o tamanho máximo em número de bytes de um ficheiro passível de inclusão no sistema de ficheiros *sofs14* é de

$$size\_max = BSLPC \cdot [N\_DIRECT + RPC \cdot (RPC + 1)] ,$$

onde *BSLPC* representa o número de bytes do conteúdo informativo de um *cluster* e *RPC* o número de referências a *clusters* que podem ser aí armazenadas.

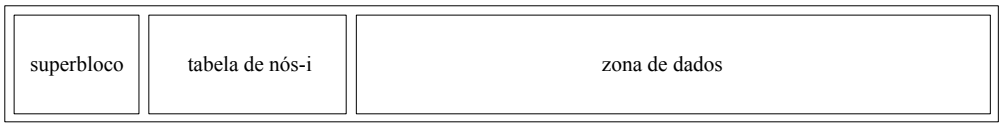
Por outro lado, a localização em memória de massa de um byte de informação, cuja posição actual no *continuum* de dados é *p*, é calculada por

- ***referenciação directa***: $0 \le l < N\_DIRECT$
  n.º do *cluster* de dados: $d[l]$ – posição dentro do *cluster*: *off*
- ***referenciação simplesmente indirecta***: $N\_DIRECT \le l < N\_DIRECT + RPC$
  n.º do *cluster* onde está localizada a continuação do *array de referências directas*: *i1* ou $i1[0]$
  conteúdo do *cluster*, entendido como um *array parcelar de referências directas*: *d'*
  posição relativa dentro do *array parcelar de referências directas*: $l' = (l - N\_DIRECT) \bmod RPC$
  n.º do *cluster* de dados: $d'[l']$ – posição dentro do *cluster*: *off*
- ***referenciação duplamente indirecta***: $N\_DIRECT + RPC \le l < N\_DIRECT + RPC \cdot (RPC + 1)$
  n.º do *cluster* onde está localizada a continuação do *array de referências indirectas*: *i2*
  conteúdo do *cluster*, entendido como um *array parcelar de referências indirectas*: *i'1*
  posição relativa dentro do *array parcelar de referências indirectas*: $l'' = (l - N\_DIRECT - RPC) \operatorname{div} RPC$
  n.º do *cluster* onde está localizada a continuação do *array de referências directas*: $i'1[l'']$
  conteúdo do *cluster*, entendido como um *array parcelar de referências directas*: *d'*
  posição relativa dentro do *array parcelar de referências directas*: $l' = (l - N\_DIRECT - RPC) \bmod RPC$
  n.º do cluster de dados: $d'[l']$ – posição dentro do cluster: *off* .

Por último, o tamanho em número de bytes do ficheiro descrito vem expresso pelo campo `size` e o número de *clusters* que o seu conteúdo informativo ocupa, pelo campo `cluCount`. Note-se, porém, que o entendimento dado a estas variáveis é muito preciso: *tamanho* significa o maior valor alguma vez assumido por *posição actual* após a escrita de um byte no *continuum* de dados; *número de clusters ocupados* corresponde ao número de *clusters* que estão efectivamente reservados para armazenamento do conteúdo informativo, incluindo portanto, se necessário, os *clusters* de armazenamento dos *arrays parcelares de referências directas e indirectas*.

Organização interna

O dispositivo de memória de massa, enquanto *array* de blocos, é entendido como estando dividido em três grandes regiões com significado bem definido



- *superbloco* – ocupa o primeiro bloco e contém informação global sobre o sistema de ficheiros, nomeadamente sobre o tamanho e a localização das regiões restantes;
- *tabela de nós-i* – constitui um *array* cujos elementos são nós-i; os nós-i *livres* formam uma lista que está organizada como um FIFO para possibilitar que, no acto de reserva de um nó-i, se tenha prioritariamente acesso a um nó-i no estado dito *limpo*;
- *zona de dados* – constitui um *array* cujos elementos são *clusters* de dados; os *clusters* de dados *livres* estão enquadrados em diferentes estruturas, que no seu conjunto formam igualmente um FIFO, para permitir que, no acto de reserva de um *cluster* de dados, se tenha prioritariamente acesso a um *cluster* de dados no estado dito *limpo*.

A ocupação completa do *array* de blocos impõe que a equação abaixo tenha soluções inteiras

$$NTBlk = 1 + NBlkInT + NTClt \cdot BLOCKS\_PER\_CLUSTER ,$$

em que *NTBlk* representa o número total de blocos do dispositivo de memória de massa, *NBlkInT* o número de blocos que a tabela de nós-i ocupa e *NTClt* o número total de *clusters* da zona de dados.

A estrutura de dados que define o *superbloco* pode considerar-se dividida em quatro grandes regiões de parametrização

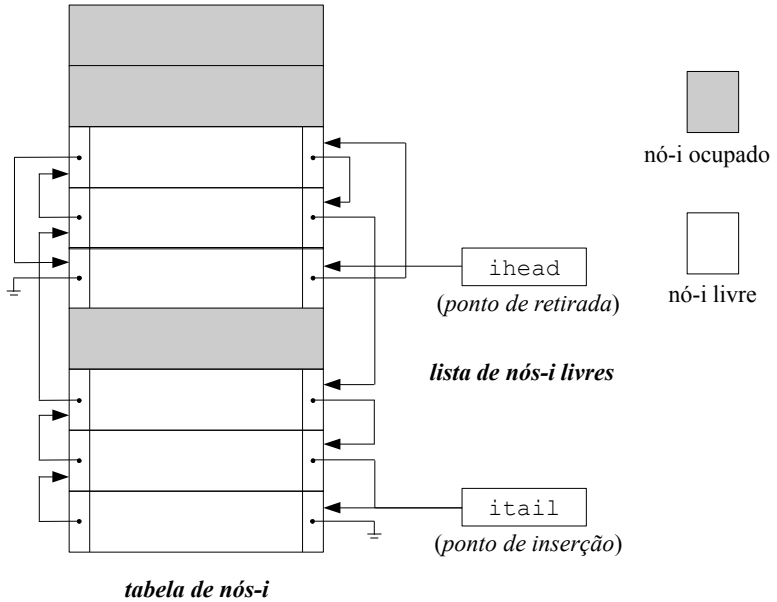
- *cabeçalho* – é formado pelos campos `magic`, `version`, `name`, `nTotal` e `mStat` que contêm o código de identificação e a versão do sistema de ficheiros, o nome e o tamanho (em número de blocos) do dispositivo de memória de massa onde ele está instalado e uma *flag* de sinalização que indica se o dispositivo foi adequadamente *desmontado* (retirado de operação) a última vez que foi *montado* (posto em operação);
- *caracterização da tabela de nós-i* – descreve a localização `iTableStart`, e o tamanho (em número de blocos ocupados, `iTableSize`, e em número de elementos, `iTotal`), da tabela de nós-i que é suposta estar organizada num *array*; os nós-i *livres*, cujo número é indicado em `iFree`, formam adicionalmente uma lista biligada cujos índices da cabeça, `iHead`, e da cauda, `iTail`, correspondentes, respectivamente, aos pontos de retirada e de inserção de elementos na lista, são também fornecidos (trata-se no fundo de uma memória de tipo FIFO dinâmica que interliga todos os nós-i presentemente livres, usando os próprios nós-i como nós da lista);
- *caracterização da zona de dados* – descreve a localização `dZoneStart`, o tamanho em número de elementos, `dZoneTotal`, da zona de dados que é suposta estar organizada num array de *clusters*; os *clusters livres*, cujo número é indicado em `dZoneFree`, estão organizados em três componentes de referência principais: as *caches* de *retirada* e de *inserção* de referências, `dZoneRetriev` e `dZoneInsert`, que constituem regiões de armazenamento temporário de acesso fácil e eficiente (são estruturas de dados estáticas residentes no superbloco) e a lista biligada que constitui o repositório geral de *clusters livres*, cujas referências da cabeça, `dHead`, e da cauda, `dTail`, correspondentes, respectivamente, aos pontos de retirada e de inserção de elementos na lista, são também fornecidos (trata-se no fundo de uma memória de tipo FIFO dinâmica que interliga todos os *clusters livres* não referenciados nas *caches*, usando os próprios *clusters* como nós da lista);

- *zona reservada* – espaço excedente não utilizado para garantir que o tamanho em bytes de da estrutura de dados `SOSuperBlock` é exactamente igual a `BLOCK_SIZE`.

```
typedef struct soSuperBlock
      {
      /* Header */
        uint32_t magic;         /* magic number - file system id number */
        uint32_t version;                              /* version number */
        unsigned char name[PARTITION_NAME_SIZE+1];       /* volume name */
        uint32_t nTotal;        /* total number of blocks in the device */
        uint32_t mStat;        /* flag signaling if the file system was
                          properly unmounted the last time it was mounted */
      /* Inode table metadata */
        uint32_t iTableStart;     /* physical number of the block where
                                              the table of inodes starts */
        uint32_t iTableSize;         /* number of blocks that the table
                                                 of inodes comprises */
        uint32_t iTotal;                      /* total number of inodes */
        uint32_t iFree;                        /* number of free inodes */
        uint32_t iHead;        /* index of the array element that forms
                              the head of the double-linked list of free
                                             inodes (point of retrieval) */
        uint32_t iTail;        /* index of the array element that forms
                              the tail of the double-linked list of free
                                             inodes (point of insertion) */
      /* Data zone metadata */
        uint32_t dZoneStart;      /* physical number of the block where
                           the data zone starts (physical number of the
                                                 first data cluster) */
        uint32_t dZoneTotal;           /* total number of data clusters */
        uint32_t dZoneFree;             /* number of free data clusters */
        struct fCNode dZoneRetriev;    /* retrieval cache of references
                                               to free data clusters */
        struct fCNode dZoneInsert;     /* insertion cache of references
                                               to free data clusters */
        uint32_t dHead;      /* logical number of the data cluster that
                  forms the head of the double-linked list of free data
                                            clusters (point of retrieval) */
        uint32_t dTail;      /* logical number of the data cluster that
                  forms the tail of the double-linked list of free data
                                            clusters (point of insertion) */
      /* Padded area to ensure superblock structure is BLOCK_SIZE
         bytes long */
        unsigned char reserved[RESERV_AREA_SIZE];
      } SOSuperBlock;
```

Aspectos operacionais

O esquema abaixo ilustra o modo como a *tabela de nós-i* está operacionalmente organizada.



Como foi referido atrás, a *zona de dados* está organizada primeiramente num *array* de *clusters* de dados. Neste sentido, uma *referência* a um *cluster* constitui o índice ou o *número lógico* do *cluster* no *array*. O *número físico* correspondente é, no fundo, o índice do primeiro bloco que o forma na visão lógica do dispositivo físico como um *array* de blocos. A relação entre ambos é dada pela equação

$$NFClt = \text{dzone\_start} + NLClt * BLOCKS\_PER\_CLUSTER \ ,$$

onde NFClt representa o número físico e NLClt o número lógico do *cluster*.

A estrutura de dados que define o *cluster* pode considerar-se dividida em duas grandes regiões

- *cabeçalho* – contém os campos `prev`, `next`, e `stat`; os dois primeiros têm a ver com o facto que os próprios *clusters*, se *livres* e incluídos no repositório geral ou *ocupado* e pertencendo ao conteúdo informativo de um ficheiro, formarem nós de estruturas dinâmicas, o terceiro especifica o estado do *cluster*: quando *ocupado*, ou *livre* no estado sujo, indica o número do nó-i a que pertence(u), quando *livre* no estado limpo é igual a NULL_INODE;
- *corpo* – constitui a região de armazenamento propriamente dita, estruturada de modo a poder conter parte do *continuum* de dados do ficheiro, um sub-*array* de referências a outros clusters ou um sub-*array* de entradas de directório.

O tipo de dados associado, `SODataClust`, é definido por

```
typedef struct soDataClust
  {
  /* Header */
    uint32_t prev;    /* if the data cluster is free and resides in
      the general repository of free data clusters or is in use and
     belongs to the information content of a file, reference to the
        previous data cluster in the double-linked list; when it is
              free and its reference is in one of the caches in the
                                superblock,reference to NULL_CLUSTER */
    uint32_t next;    /* if the data cluster is free and resides in
      the general repository of free data clusters or is in use and
     belongs to the information content of a file, reference to the
            next data cluster in the double-linked list; when it is
              free and its reference is in one of the caches in the
                                superblock,reference to NULL_CLUSTER */
    uint32_t stat;                    /* status of the data cluster */
  /* Body */
    union infoContent info;          /* cluster information content */
  } SODataClust;
```

Em princípio, quando o tamanho de um ficheiro cresce, deve ser usado um *cluster* de dados no estado *limpo* para armazenar a nova informação; porém, se todos os *clusters* de dados livres estiverem *sujos*, aquele que for usado tem que ser colocado em primeiro lugar no estado *limpo*, o que significa que a parte do conteúdo informativo do ficheiro apagado, nele contido, não poderá mais vir a ser recuperado.

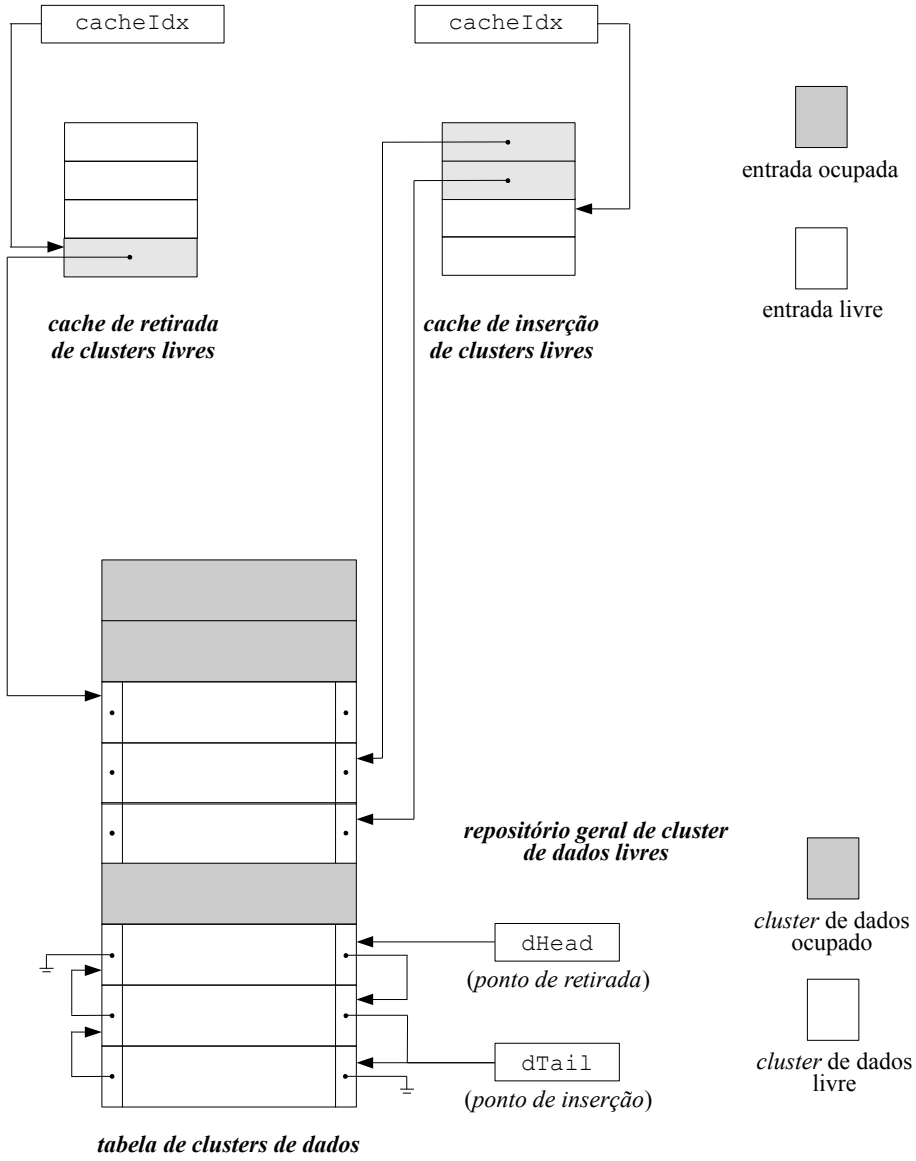
A região de armazenamento de um *cluster* é descrita pelo tipo de dados **union** `infoContent` para permitir a versatilidade acima referida.

```
union infoContent
{
  unsigned char data[BSLPC];                          /* byte stream */
  uint32_t ref[RPC];        /* sub-array of data cluster references */
  SODirEntry de[DPC];            /* sub-array of directory entries */
};
```

As *caches* de *retirada* e de *inserção* de referências a *clusters* de dados são baseadas no tipo de dados seguinte

```
struct fCNode
{
  uint32_t cacheIdx;              /* index of the first filled/free
                                                        array element */
  uint32_t cache[DZONE_CACHE_SIZE];  /* storage area whose elements
                       are the logical numbers of free data clusters */
};
```

O esquema abaixo ilustra o modo como a *tabela de clusters* de dados está operacionalmente organizada.



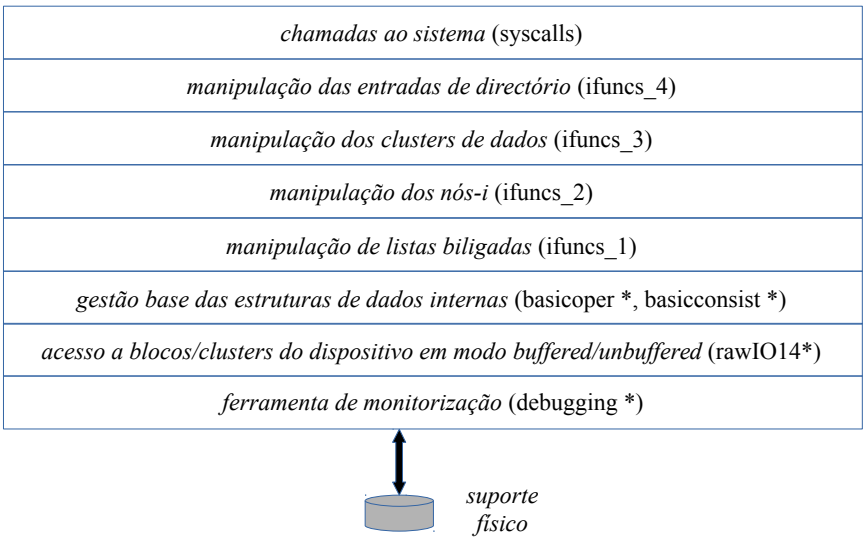
Implementação

Dada a sua complexidade, a implementação do sistema de ficheiros *sofs14* está organizada numa estrutura hierárquica de funcionalidades que pressupõe diferentes níveis de abstracção. Este tipo de decomposição de soluções é comummente conhecido como *arquitectura em camadas*.

Cada *camada*, ou nível de abstracção, está organizada num ou mais módulos e apresenta ao programador um *API* (*Application Programming Interface*) que descreve sintáctica e semanticamente as operações que podem ser efectuadas a este nível. Idealmente, cada camada deve comunicar apenas com a camada imediatamente abaixo. Contudo, como em muitos casos esta

regra exigiria a replicação de operações de camadas inferiores em camadas superiores, é aceitável que cada camada possa comunicar com todas aquelas que lhe são inferiores.

Apresenta-se a seguir a arquitectura de camadas da implementação de *sofs14*.



As camadas referenciadas com um '*' são fornecidas já implementadas.

O objectivo do trabalho será, pois, a implementação das restantes camadas e de um programa que permita a formatação de um dispositivo com o sistema de ficheiros *sofs14*.

A validação da implementação será feita através da integração do sistema de ficheiros numa plataforma hardware que execute o sistema de operação Linux.

São ainda fornecidas algumas ferramentas para apoio ao teste e validação do código das diferentes camadas à medida que este é desenvolvido.