SISTEMA DE FICHEIROS sofs15

Nota prévia

sofs15 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 2015/2016.

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

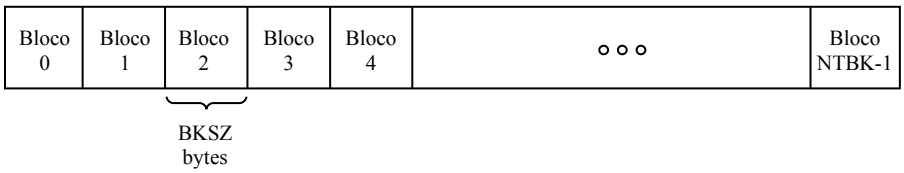
O que é um sistema de ficheiros ?

Quase todas os programas, durante a sua execução, produzem, consultam e/ou alteram quantidades variáveis de informação que é armazenada de um modo mais ou menos permanente em dispositivos físicos agrupados sob o nome genérico de *memória de massa*.

Um conjunto variado de requisitos são impostos a este tipo de funcionalidade

- *ser não volátil* – a informação deve poder existir antes da criação do(s) processo(s) que a vai(ão) usar, e sobreviver à sua terminação, mesmo quando o sistema computacional é desligado;
- *ter uma grande capacidade de armazenamento* – a informação manipulada pelos processos pode ultrapassar em muito aquela que é directamente armazenada nos espaços de endereçamento próprios;
- *providenciar um acesso eficiente* – o acesso a informação específica deve ser efectuado do modo mais simples e rápido possível;
- *manter a integridade da informação* – a informação armazenada deve estar protegida contra a alteração e a corrupção acidentais;
- *permitir a partilha da informação* – a informação deve ser acessível concorrentemente a múltiplos processos que fazem uso dela.

Os discos magnéticos representam sem dúvida, ainda hoje em dia, o tipo de dispositivos mais usados na materialização do armazenamento *on-line* de informação. As razões da popularidade prendem-se com a grande capacidade de armazenamento, a fiabilidade associada à realização de um número extremamente elevado de operações de leitura e escrita e o seu baixo preço. Estruturalmente, verifica-se que



- o dispositivo pode ser encarado de um modo lógico como um *array* formado por *NTBK* blocos, cada um deles contendo *BKSZ* bytes (tipicamente, *BKSZ* varia entre 256 e 8K);
- o acesso a um bloco particular, para leitura ou escrita, é feito na sua globalidade, fornecendo o seu número identificador, e o tempo associado não depende do tipo de operação.

A manipulação da informação contida directamente no dispositivo físico não pode, porém, ser deixada à responsabilidade do programador de aplicações. A complexidade inerente à sua estrutura interna e a necessidade de garantir critérios de qualidade, relacionados com a eficiência no acesso e a integridade e a partilha, exigem a criação de um modelo uniforme de interacção.

O conceito de *ficheiro* surge, assim, como a *unidade lógica de armazenamento em memória de massa e de acesso à informação*. Quer com isto dizer-se que a leitura e a escrita de dados em memória de massa se faz sempre no âmbito estrito de um ficheiro.

Um *ficheiro* apresenta como elementos básicos

- *o nome* – a forma genérica de referenciar uma informação particular;
- *o conteúdo informativo* – a informação propriamente dita, organizada numa sequência de bits, bytes, linhas ou registos, cujo formato preciso é definido pelo criador do ficheiro e que tem que ser conhecido por quem a ele acede.

Sob o ponto de vista do programador de aplicações, um ficheiro pode ser visto como um tipo de dados específico, caracterizado por um conjunto de *atributos* e por um conjunto de *operações*.

O papel do sistema de operação é implementar este tipo de dados, fornecendo um leque alargado de *chamadas ao sistema* que estabelecem um interface simples e seguro de comunicação com a memória de massa. A parte do sistema de operação que se dedica a esta tarefa, designa-se de *sistema de ficheiros*. Diferentes implementações conduzem a diferentes tipos de sistemas de ficheiros.

Os sistemas de operação actuais suportam diferentes sistemas de ficheiros, associados, quer com dispositivos físicos distintos, quer com o mesmo dispositivo. Este último aspecto facilita a chamada *interoperacionalidade*, estabelecendo um meio comum de partilha de informação entre sistemas ditos heterogéneos.

O ficheiro como um tipo de dados

Os *atributos* de um ficheiro são variados e dependem da implementação. Destaca-se aqui um conjunto mínimo que está habitualmente presente

- *nome* – o ficheiro passa a ter uma identidade própria com a atribuição de um *nome*, tornando-se independente do processo e do utilizador que o criaram, e mesmo do sistema de operação em que foi criado;
- *identificador interno* – o *nome*, enquanto elemento individualizador de um ficheiro, é adequado para uma referência externa (tipicamente de origem humana), mas é pouco prático em termos de organização interna; o *identificador interno* constitui, por isso, o elemento de discriminação que vai permitir referenciar os outros atributos do ficheiro e, através deles, a informação propriamente dita;
- *tamanho* – o comprimento do conteúdo informativo, normalmente em número de bytes;
- *pertença* – a indicação de quem criou o ficheiro e, portanto, a quem ele pertence;
- *protecção* – o controlo de acesso, especificando quem pode ler o seu conteúdo, escrever nele ou realizar operações de execução;
- *monitorização de acesso* – datas e tempos dos instantes de criação, de última modificação e de último acesso, por exemplo;
- *localização* – lista ordenada com a identificação dos blocos (tipicamente, os índices dos elementos do *array* de blocos que constitui a visão lógica do dispositivo) que contêm o seu conteúdo informativo;
- *tipo* – os sistemas de ficheiros admitem ficheiros de tipos diversos
  - *ficheiros regulares* – são os ficheiros convencionais, contêm todo o tipo de informação que é normalmente armazenada em memória de massa;
  - *directórios* – são ficheiros internos, com um formato pré-definido, que descrevem a estrutura hierárquica da organização subjacente; os dispositivos físicos de armazenamento de massa contemporâneos podem conter em condições normais dezenas de milhar, ou milhões, de ficheiros regulares, se todos eles estivessem colocados ao mesmo nível, a referência a um deles em particular, ou a atribuição de um nome a um novo ficheiro, tornar-se-iam pela sua complexidade operações quase impossíveis de realizar na prática;
  - *atalhos* – são ficheiros internos, com formato pré-definido, que descrevem a localização de um outro ficheiro através da especificação do seu encaminhamento na estrutura hierárquica pré-existente
  - *ficheiros especiais* – alguns sistemas operação fornecem uma visão integrada das operações de entrada e de saída de dados, usando o mesmo conjunto de operações, quer para acesso a ficheiros, quer para acesso aos dispositivos físicos; quando existem, os *ficheiros especiais* modelam esses dispositivos.

As *operações* que se podem realizar sobre um ficheiro são igualmente variadas e dependem do sistema de operação. Há, porém, um núcleo base que de algum modo está sempre presente.

Operações realizadas sobre *ficheiros regulares*

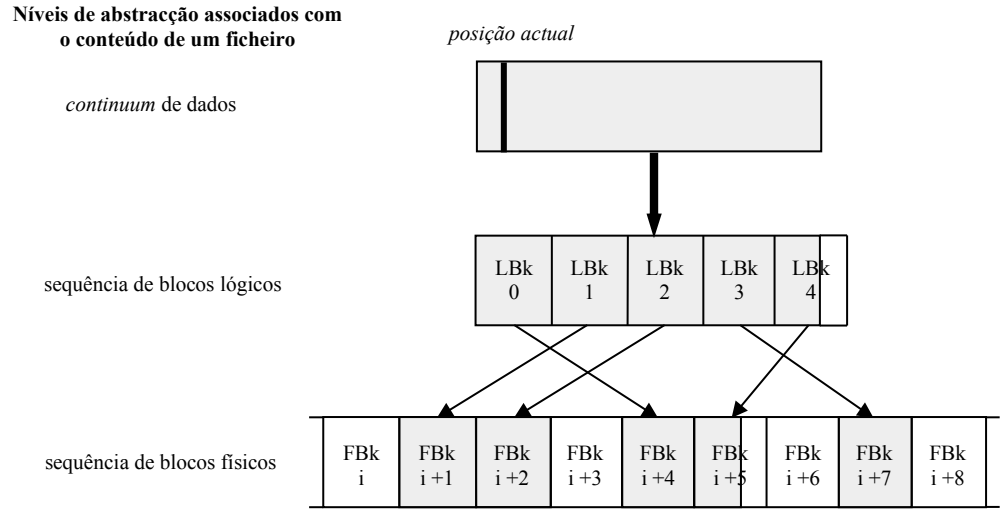
- *criação* – é criada no directório actual uma *entrada* (referência) para o ficheiro; a entrada vai conter o *nome* e o *identificador interno*, os atributos restantes são inicializados, nalguns casos usam-se valores passados como parâmetros da operação; o *tamanho* do ficheiro, em particular, é colocado a zero, sinalizando que está vazio;
- *abertura* – é a operação alternativa à *criação*, sendo efectuada sempre que o ficheiro referenciado já existe; as entradas do directório são pesquisadas para localizar a entrada respectiva, o *identificador interno* é a seguir usado para aceder aos restantes atributos;

As operações de *criação* e de *abertura* efectuam-se sempre que se pretende aceder ao conteúdo informativo de um ficheiro. O objectivo é construir em memória principal uma estrutura de dados que viabilize uma comunicação eficiente durante as operações subsequentes efectuadas sobre o seu conteúdo informativo.

- *fecho* – é a operação complementar das anteriores, traduz a terminação da comunicação com o ficheiro; o bloco de dados relativo à parte do conteúdo que estava a ser referenciado na altura, é escrito no dispositivo, se tiver ocorrido alteração; os atributos relevantes são actualizados com a informação da estrutura de dados residente em memória principal e o espaço ocupado por ela é libertado;
- *posicionamento* – o conteúdo do ficheiro é entendido na perspectiva do programador de aplicações como um *continuum* de dados; o local nesse *continuum* onde irá ocorrer a próxima transferência de informação, pode ser definido através da operação de posicionamento que fixa a chamada *posição actual*;

Quando um ficheiro é *criado* ou *aberto*, *posição actual* aponta para o início desse *continuum*; a realização subsequente de operações de *escrita* ou de *leitura* conduzem à actualização do valor de *posição actual*, que passa a apontar para a posição imediatamente a seguir à localização de escrita ou de leitura do último valor.

- *escrita* – é a operação de transferência de informação *para o* ficheiro; os dados são escritos a partir da *posição actual* e retirados de uma região do espaço de endereçamento do processo através de uma variável passada como parâmetro da operação;
- *leitura* – é a operação de transferência de informação *do* ficheiro; os dados são lidos a partir da *posição actual* e armazenados numa região do espaço de endereçamento do processo através de uma variável passada como parâmetro da operação;



A transposição da abstracção do *continuum* de dados para o armazenamento efectivo desses dados em blocos do dispositivo físico impõe a tradução da *posição actual* em dois valores distintos: um *bloco específico* e um *deslocamento* dentro desse bloco. Assim, as operações de *escrita* e de *leitura* supõem a monitorização constante do bloco onde são efectuadas: quando se iniciam, é necessário garantir que uma cópia do bloco está acessível na estrutura de dados residente em memória principal; depois, durante a realização das operações, se os limites do bloco forem ultrapassados, o bloco em causa é escrito no dispositivo, caso tenha ocorrido uma alteração, e o bloco seguinte é lido para a estrutura de dados residente em memória principal.

- *apagamento* – o ficheiro é retirado do sistema de ficheiros; os blocos de dados de armazenamento do seu conteúdo informativo são libertados e a entrada do directório correspondente, bem como a região onde estão armazenados os restantes atributos, são sinalizadas vazias; note-se, porém, que não pode haver comunicação estabelecida com o ficheiro quando a operação é efectuada.

Operações realizadas sobre *directórios*

- *criação* – é criada no directório actual uma *entrada* (referência) para o directório; a entrada vai conter o *nome* e o *identificador interno*, os atributos restantes são inicializados, nalguns casos usam-se valores passados como parâmetros da operação; ao contrário do que se passa com um *ficheiro regular*, o *tamanho* do directório não é colocado a zero, um directório vazio contém sempre referências a si próprio e ao directório hierarquicamente imediatamente acima para que a navegação na *árvore* de directórios possa ser implementada de um modo eficiente;
- *apagamento* – o directório é retirado do sistema de ficheiros; tal como para um *ficheiro regular*, os blocos de dados de armazenamento do seu conteúdo informativo são libertados e a entrada do directório correspondente, bem como a região onde estão armazenados os restantes atributos, são sinalizadas vazias; o directório tem que estar vazio para que a operação possa ter lugar.

Operações realizadas sobre *atalhos*

- *criação* – é criada no directório actual uma *entrada* (referência) para o atalho; a entrada vai conter o *nome* e o *identificador interno*, os atributos restantes são inicializados, nalguns casos usam-se valores passados como parâmetros da operação; o *tamanho* do atalho é colocado num valor que corresponde ao tamanho do *string* que define o encaminhamento e o conteúdo informativo contém esse encaminhamento;
- *apagamento* – o atalho é retirado do sistema de ficheiros; tal como para um *ficheiro regular*, o bloco de dados de armazenamento do seu conteúdo informativo é libertado e a entrada do directório correspondente, bem como a região onde estão armazenados os restantes atributos, são sinalizadas vazias.

Operações realizadas sobre ficheiros genéricos (*ficheiros regulares*, *directórios* e *atalhos*)

- *consulta dos atributos* – permite o acesso à generalidade dos atributos de um ficheiro;
- *alteração dos atributos* – permite a modificação de alguns dos atributos de um ficheiro após a sua criação, exemplos significativos são a *pertença* e a *protecção*;
- *alteração do nome* – o nome do ficheiro é modificado; algumas implementações permitem ainda o seu deslocamento dentro da estrutura hierárquica pré-existente, isto é, a referência ao ficheiro pode ser deslocada de um directório para outro;

Estas operações são tipicamente efectuadas apenas pelo processo lançado pelo utilizador a quem o ficheiro actualmente pertence. A operação de *alteração do nome*, por outro lado, exige que não haja comunicação estabelecida com o ficheiro para poder ser realizada.

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 e que exige um conhecimento profundo das estruturas de dados internas e da funcionalidade apresentada pelo sistema de operação-alvo, estando, por isso, só estão ao alcance de programadores de sistemas experientes.

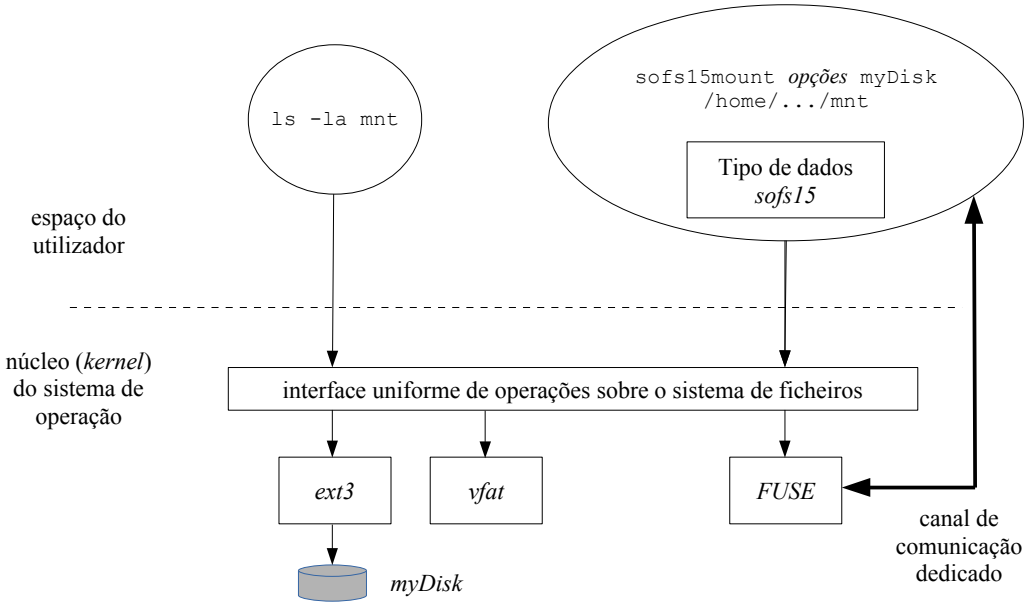
FUSE (*File system in User SpacE*) constitui uma solução alternativa muito engenhosa que visa a construção de sistemas de ficheiros no espaço do utilizador, como se se tratasse de meras aplicações, impedindo que eventuais inconsistências e erros que surjam na sua implementação, sejam transmitidos directamente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

A infraestrutura oferecida é formada por duas partes principais:

- *módulo de interface com o sistema de ficheiros* – funciona como um mediador de comunicações entre o interface uniforme de operações sobre ficheiros fornecido pelo *kernel* e a respectiva implementação em espaço do 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 *sofs15*, 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 *ficheiro* (neste caso, *sofs15*). 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 *sofs15mount*, 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 *sofs15mount*;
- 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 *sofs15mount*. Usando a tabela `struct fuse_operations op`, a aplicação converte a operação em uma ou mais operações do tipo de dados *sofs15*, 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 sofs15

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 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 materializar 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 tem um tamanho variável.

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`'.

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, *atalho*, *directório* ou *ficheiro regular*, 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*.

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 através 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 seguro 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 deslocação 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 prev;      /* if inode is free, reference to the previous
                      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 next;    /* if inode is free, reference to the next 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.prev` e `vD2.next`, 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 formar uma lista ordenada de referências aos *clusters* da unidade de memória de massa aonde a informação está armazenada. Tudo se passa como se o *continuum* fosse dividido em pedaços de tamanho de um *cluster* e a localização de cada *cluster*, correspondente ao índice do seu primeiro bloco constituinte quando o dispositivo físico é visto como um *array* de blocos, 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

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

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

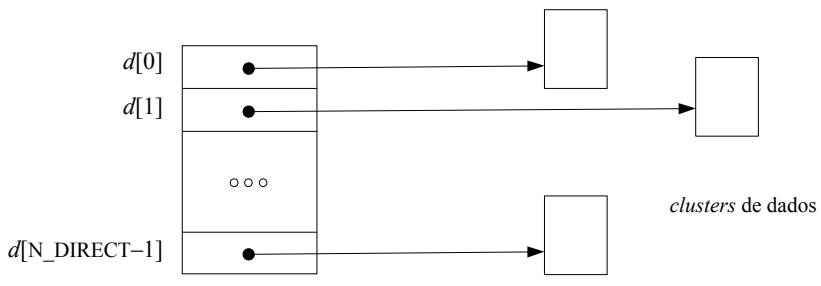
$$off = p \text{ mod } BSLPC$$

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

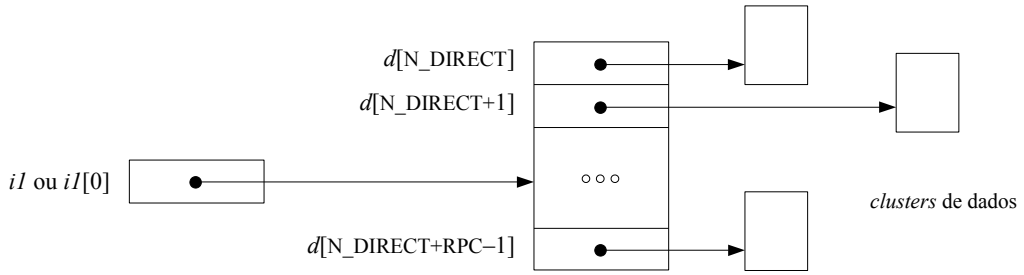
Assim sendo, o nó-i descritivo de um ficheiro terá que incluir de algum modo o *array d*. Contudo, se se pretender que o sistema de ficheiros possa conter ficheiros muito grandes, o tamanho desse *array* pode assumir um tamanho incomportável. No caso presente em que *BSLPC* é igual a 2Kbytes, a possibilidade de se ter um ficheiro com um conteúdo informativo de 2Gbytes implicaria que o número de elementos do *array d* fosse 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 referência indirecta de níveis progressivamente mais elevados.

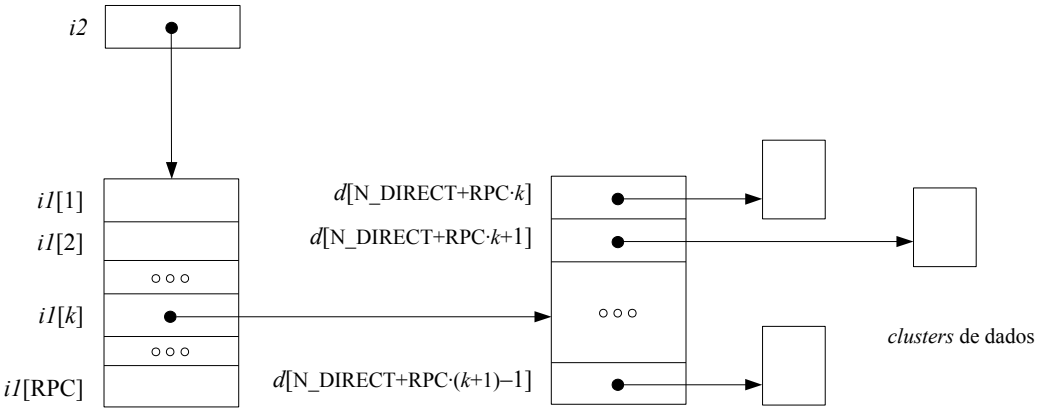
Referenciação directa



Referenciação simplesmente indirecta



Referenciação duplamente indirecta



Neste caso em que foram implementados mecanismos de referência 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 *sofs15* 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.

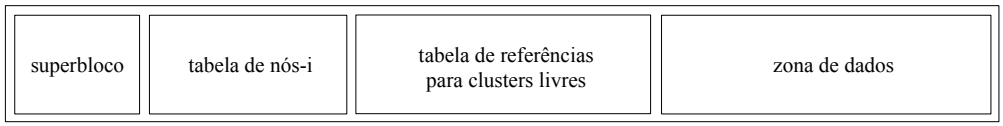
Por outro lado, a localização em memória de massa de um byte de informação, cuja posição actual no *continuum* de dados é *p*, é calculada por

- ***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 quatro 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 adicionalmente uma lista biligada que está organizada como um FIFO circular;
- *tabela de referências para clusters livres* – constitui um *array* cujos elementos são referências para os *clusters* da zona de dados que estão correntemente livres, está organizada como um FIFO linear numa implementação estática;
- *zona de dados* – constitui um *array* cujos elementos são os *clusters* de dados.

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

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

em que *NTBlk* representa o número total de blocos do dispositivo de memória de massa, *NBlkInT* e *NBlkFCT*, respectivamente, o número de blocos que a tabela de nós-i e que a tabela de referências para *clusters* livres ocupam 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

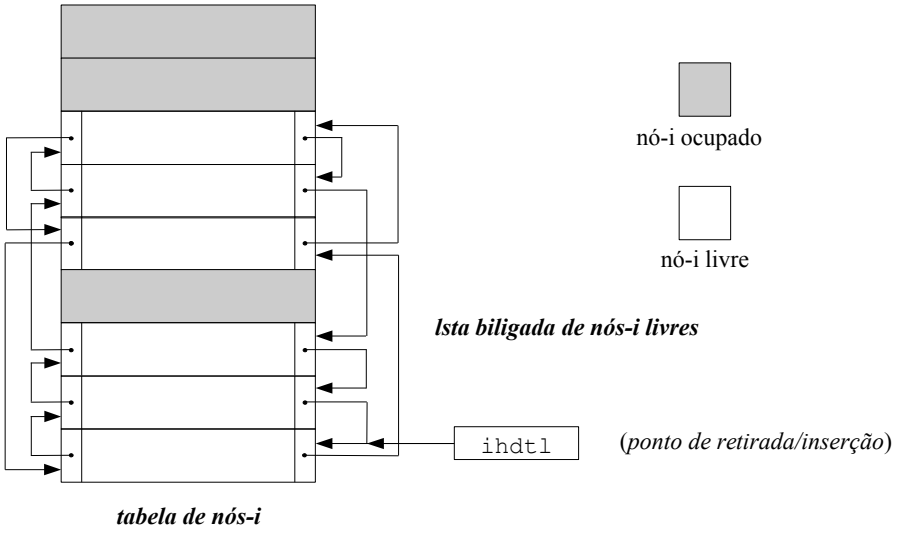
- *cabeçalho* – é formado pelos campos `magic`, `version`, `name`, `ntotal` e `mstat` que contêm o código de identificação e a versão do sistema de ficheiros, o nome e o tamanho (em número de blocos) do dispositivo de memória de massa onde ele está instalado e uma *flag* de sinalização que indica se 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 `itable_start`, e o tamanho (em número de blocos ocupados, `itable_size`, 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 circular cujo índice da cabeça / cauda, `ihdtl`, correspondente simultaneamente ao ponto de retirada e de inserção de elementos na lista, é também fornecido (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, `dzone_start`, e o tamanho em número de elementos, `dzone_total`, da zona de dados que é suposta estar organizada num *array* de *clusters*; os *clusters livres*, cujo número é indicado em `dzone_free`, estão organizados em três componentes de referência principais: as *caches* de *retirada* e de *inserção* de referências, `dzone_retriev` e `dzone_insert`, que constituem regiões de armazenamento temporário de acesso rápido e eficiente (são estruturas de dados estáticas residentes no superbloco) e a tabela de referências para *clusters* livres descrita pela sua localização, `tbfreeclust_start`, e tamanho em número de blocos, `tbfreeclust_size`, e pelos pontos de retirada e de inserção de elementos na lista, `tbfreeclust_head` e `tbfreeclust_tail` (trata-se no fundo de uma memória de tipo FIFO linear de implementação estática que contém todas as referências a *clusters* livres não presentes nas *caches*);

- *zona reservada* – espaço excedente para garantir que o tamanho em bytes de `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 itable_start;    /* physical number of the block where
                                        the table of inodes starts */
        uint32_t itable_size;        /* 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 ihdtl;        /* index of the array element that forms
                        the head/tail of the double-linked list of free
                                inodes (point of retrieval/insertion) */
      /* Data zone metadata */
        uint32_t dzone_start;     /* physical number of the block where
                           the data zone starts (physical number of the
                                                 first data cluster) */
        uint32_t dzone_total;          /* total number of data clusters */
        uint32_t dzone_free;           /* number of free data clusters */
        struct fCNode dzone_retriev;   /* retrieval cache of references
                                              to free data clusters */
        struct fCNode dzone_insert;    /* insertion cache of references
                                              to free data clusters */
        uint32_t tbfreeclust_start;     /* physical number of the block
                                   where the table of references to free
                                                data clusters starts */
        uint32_t tbfreeclust_size;   /* number of blocks that the table
                                of references to free data clusters comprises */
        uint32_t tbfreeclust_head;   /* index of the array element that
                                forms the head of the table of references to
                                     free data clusters (point of retrieval) */
        uint32_t tbfreeclust_tail;   /* index of the array element that
                                forms the tail of the table of references to
                                     free data clusters (point of insertion) */
      /* Padded area to ensure superblock structure is BLOCK_SIZE
         bytes long */
        unsigned char reserved[BLOCK_SIZE - PARTITION_NAME_SIZE - 1
                  - 16 * sizeof(uint32_t) - 2 * sizeof(struct fCNode)];
      } 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 caracteriza um *cluster* pode considerar-se estruturada de modo a poder conter alternativamente parte do *continuum* de dados do ficheiro, um sub-*array* de referências a outros clusters ou um sub-*array* de entradas de directório.

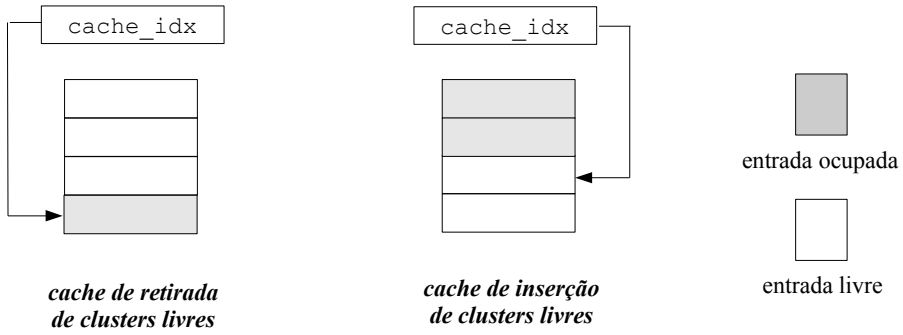
O tipo de dados associado, `SODataClust`, é definido por

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

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

```
struct fCNode
{
  uint32_t cache_idx;              /* 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 referência a clusters livres está organizada.



cache de retirada de clusters livres

cache de inserção de clusters livres

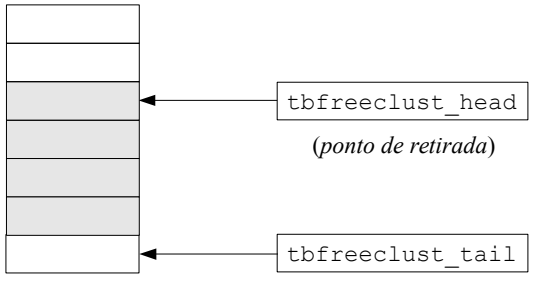


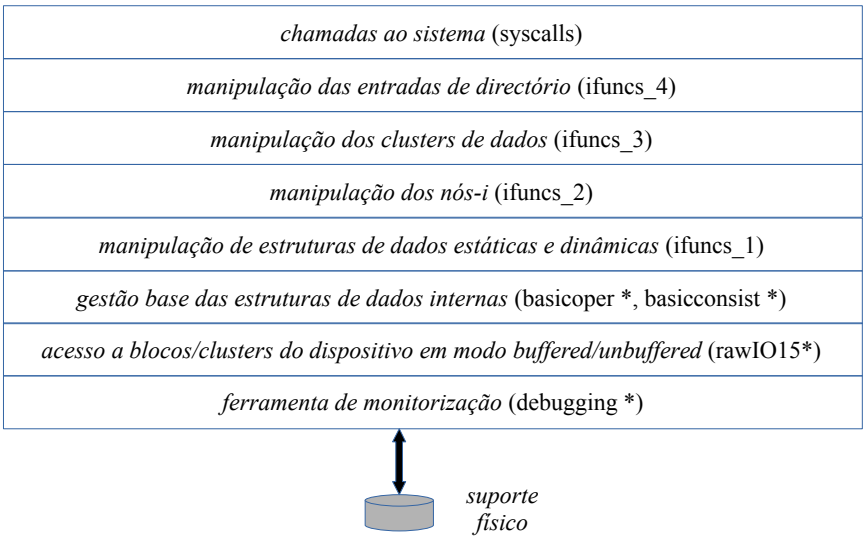
tabela de referências para clusters livres

Implementação

Dada a sua complexidade, a implementação do sistema de ficheiros *sofs15* 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 *sofs15*.



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 *sofs15*.

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.