2 – Systèmes de fichiers
Partie 3 : Interface utilisateur

HEPIA
Année académique 2015/2016

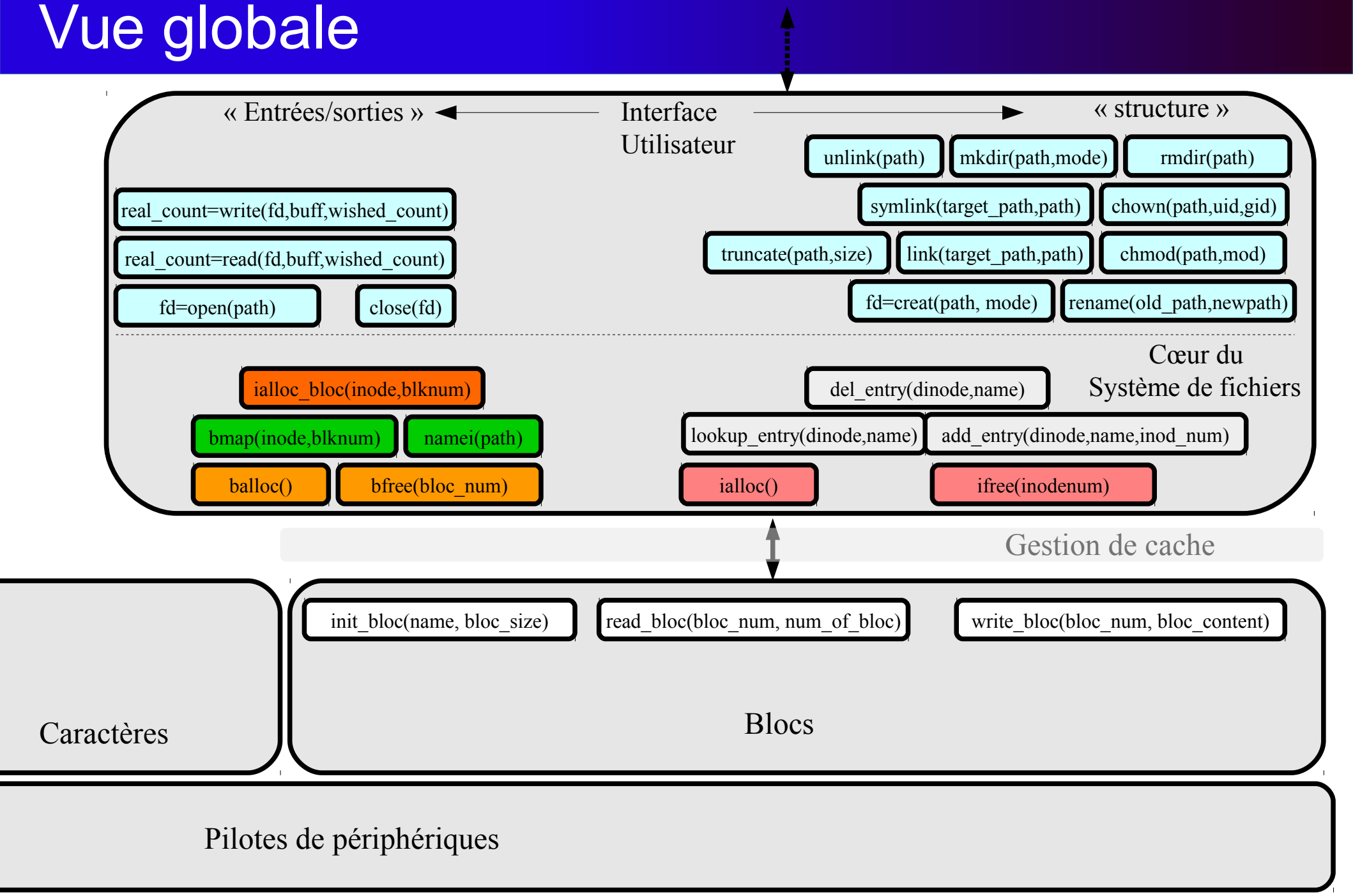
Contenu

Introduction et rappels

Opérations sur la structure du système de fichiers

Opérations sur d'entrées/sorties sur les fichiers

Vue globale


« Entrées/sorties »
Interface Utilisateur
« structure »
unlink(path)
mkdir(path,mode)
rmdir(path)
real_count=write(fd,buff,wished_count)
symlink(target_path,path)
chown(path,uid,gid)
real_count=read(fd,buff,wished_count)
truncate(path,size)
link(target_path,path)
chmod(path,mod)
fd=open(path)
close(fd)
fd=creat(path, mode)
rename(old_path,newpath)
Cœur du Système de fichiers
ialloc_bloc(inode,blknum)
del_entry(dinode,name)
bmap(inode,blknum)
namei(path)
lookup_entry(dinode,name)
add_entry(dinode,name,inod_num)
balloc()
bfree(bloc_num)
ialloc()
ifree(inodenum)
Gestion de cache
init_bloc(name, bloc_size)
read_bloc(bloc_num, num_of_bloc)
write_bloc(bloc_num, bloc_content)
Caractères
Blocs
Pilotes de périphériques

Introduction

Attention : par soucis de simplicité, les fonctions de l'interface qui suivent :

- Ne gèrent pas un accès concurrent au système de fichier.

- Ne traitent pas les flags d'accès aux fichiers (O_APPEND, O_CREAT, etc).

- Ne traitent ni les cas d'erreurs, ni les gestions d'autorités d'accès.

- Supposent que les arguments qui sont des chemins d'accès sont toujours absolus, les chemins relatifs au répertoire de travail du processus ne sont pas traités.

- Supposent que les changements effectués sur les structures de données seront reflétés correctement sur les données stockées sur le disque (table des inodes,bitmaps des blocs/inodes libres) avant un nouvel appel à d'autres fonctions de l'interface :

  Exemple : un appel à `read()` précédé d'un `write()` verra immédiatement le changement effectué par `write()`.

Contenu

Introduction et rappels

Opérations sur la structure du système de fichiers

Opérations sur d'entrées/sorties sur les fichiers

Les fonctions truncate, chmod, chown

truncate(path,size) | chmod(path,mode) | chown(path,uid,gid)

Elle changent simplement les propriétées correspondantes dans l'inode associé au chemin indiqué en paramètre.

Exemple pour truncate :

```
truncate(path, size) :

      inode=namei(path)
      inode.size =size
```

Diminuer le champs size de l'inode équivaut à effacer toute ce qui se trouve après l'offset size dans le fichier, même si le contenu n'est pas réellement effacé.

De même, augmenter size sans toucher au contenu équivaut à créer un fichier à trou : les blocs ne sont pas alloués sur le disque, mais la lecture indiquera des octets à '\0'

La fonction unlink

unlink(path)

- N'est censé fonctionner que si le type de fichier n'est pas un répertoire

- On extrait le nom du fichier `f` du path (basename)

- Puis on trouve l'inode `dinode` du répertoire dans lequel se trouve le fichier (dirname puis namei)

- On appelle `del_entry (di,f)`

```
unlink(path) :

      dinode=namei(dirname(path))
      del_entry(dinode,basename(path))
```

La fonction link

link(target_path,linkname)

- N'est censé fonctionner que si les types de fichier ne sont pas des répertoires.

- On extrait le nom du fichier `f` de linkname (basename).

- On trouve l'inode `dinode` du répertoire dans lequel se trouve le fichier f (dirname puis namei).

- On trouve le numéro d'inode `ti` de targetpath.

- On appelle `add_entry (dinode,f,ti)` et on incrémente le nombre de liens sur de `ti`

```
link(target_path,linkname) :

      dinode=namei(dirname(linkname))
      ti=namei(targetpath)
      add_entry(dinode,basename(linkname),ti)
```

La fonction symlink

symlink(target_path,linkname)

- Crée un fichier de type SYMLINK nommé linkname dont le contenu texte est target_path

```
symlink(target_path,linkname) :

     dinode = namei(dirname(target_path)

     ni=ialloc()
     ni.i_mode |= I_SYMLINK
     ni.i_size = len(target_path)

     add_entry(dinode,basename(linkname),ni)

     fd=open(linkname)
     write(fd,target_path,len(target_path))
     close(fd)
```

- Qu'est-ce qui manque encore pour que l'accès au fichier pointé par le lien symbolique se fasse de manière transparente avec open()/read()/write() ?

La fonction rename

rename(old_path,new_path)

Elle appelle link(old_path,new_path)

Puis appelle unlink(old_path)

La fonction mkdir(1/2)

mkdir(path,mode)

Crée un fichier de type DIRECTORY avec deux entrées '.' et '..' dans le répertoire indiqué en paramètre

- On extrait le nom du répertoire `d` de path (basename).

- On trouve l'inode `dinode` du répertoire dans lequel se trouve le nouveau répertoire `d` (dirname puis namei).

- On alloue une nouvelle inode `ni` pour `d`. Puis on remplit les champs de l'inode correctement :

```
i_mode|=I_DIRECTORY ; i_size = 2*16 ; n_links=2 ;  ....
```

- Ajouter les deux entrées '.' et '..' :

```
add_entry(ni,'.',ni) ; add_entry(ni,'..',dinode)
```

- On appelle `add_entry (dinode,d,ni)`

La fonction mkdir(2/2)

```
mkdir(path,mode) :

     dinode = namei(dirname(path)

     ni=ialloc()
     ni.i_mode |= mode | I_DIRECTORY
     ni.i_size = 32
     ni.n_links=2
     ni.i_uid=...
     ni.i_guid=...

     add_entry(ni,'.', ni)
     add_entry(ni,'..',dinode)
     add_entry(dinode,basename(path),ni)
```

La fonction rmdir

rmdir(path)

L'équivalent de unlink, mais pour les répertoires.

On doit d'abord vérifier que le répertoire est vide :

- Pourquoi ?

- Indice : C'est pour la même raison que unlink ne fonctionne pas sur Des répertoires.

La fonction creat

fd=creat(path,mode)

- Crée un fichier vide de type REGULAR nommé path
- Renvoyer un descripteur de fichier

```
creat(path,mode) :

     dinode = namei(dirname(path)

     ni=ialloc()
     ni.i_mode |= mode | I_REGULAR
     ni.i_size = 0

     add_entry(dinode,basename(path),ni)

     return open(path)
```

Contenu

Introduction et rappels

Opérations sur la structure du système de fichiers

Opérations sur d'entrées/sorties sur les fichiers

Les fonctions open et close

fd=open(path)

Pourquoi une fonction open plutôt qu'une lecture directe sur le numéro d'inode ou bien sur un chemin d'accès indiqué en paramètre ?

```
open(path) :

     inode=namei(path)
     new_open_file.inode = inode
     new_open_file.current_offset=0
     fd_index=find_free_fd(file_table)
     file_table[fd_index]=new_open_file
     return fd_index
```

```
close(fd) :
     file_table[fd]=AVAILABLE
```

La fonction read

```
real_count=read(fd,buffer,wished_count)
```

Cette fonction permet d'accéder **en mode octet** à un fichier qui est stocké en mode bloc. `real_count` et `wished count` sont exprimés en octets. `wished_count` est la quantité d'octet qu'on désire recevoir dans buffer après Un appel à read, `real_count` est la quantité effectivement lue.

L'accès contigu aux blocs est assuré par la fonction bmap(), mais il faut considérer les scénarios suivants :

1. Lecture qui s'étale sur plusieurs blocs

2. Lecture inférieure à la taille d'un bloc

De plus, la lecture ne doit pas aller plus loin que la taille du fichier :

La fonction read


```
real_count read(fd,buffer,wished_count) :

    inode = file_table[fd].inode
    file_offset = file_table[fd].current_offset
    total_left = 0 si file_offset > inode.size  sinon min(inode.size-file_offset,wished_count)
    buffer_offset = 0

    Tant que total_left > 0 :
        bloc_num = bmap(inode, int(file_offset/BLOCK_SIZE))
        offset_in_bloc = file_offset % BLOCK_SIZE
        to_read = min(BLOCK_SIZE - offset_in_bloc, total_left)

        si bloc_num!= 0 :
            blk = read_bloc(bloc_num,1)
            copy_to_buffer(buffer[buffer_offset], blk[offset_in_bloc:offset_in_bloc+to_read])
        sinon
            copy_to_buffer(buffer[buffer_offset], to_read*'\0']

        file_offset + = to_read
        buffer_offset += to_read
        total_left - = to_read

    file_table[fd].current_offset=file_offset
    return buffer_offset
```

Principe de l'algorithme : à chaque itération on traque le nombre d'octets total qui reste à lire, on calcule le nombre d'octets à lire, compris entre 1 et BLOCKSIZE puis on l'ôte du nombre total d'octet à lire.

La fonction write

```
real_count=write(fd,buffer,wished_count)
```

Même principe de read() :

À chaque itération on traque le nombre d'octets total qui reste à **écrire**, on calcule le nombre d'octets à **écrire**, compris entre 1 et BLOCKSIZE puis on l'ôte du nombre total d'octet à écrire.

À la différence que :

- On peut écrire au delà de la taille du fichier. Si c'est le cas il faut mettre la taille de l'inode à jour.

- L'accès contigu aux blocs est assuré par la fonction ialloc_bloc()

La fonction write

```
real_count write(fd,buffer,wished_count) :

    inode = file_table[fd].inode
    file_offset = file_table[fd].current_offset
    total_left = wished_count
    buffer_offset = 0

    Tant que total_left > 0 :

        bloc_num = ialloc_bloc(inode, int(file_offset/BLOCK_SIZE))
        offset_in_bloc = file_offset % BLOCK_SIZE
        to_write = min(BLOCK_SIZE - offset_in_bloc, total_left)


        blk = read_bloc(bloc_num,1)
        copy_from_buffer(buffer[buffer_offset], blk[offset_in_bloc:offset_in_bloc+to_write])
        write_bloc(bloc_num,blk)

        file_offset + = to_write

        si file_offset > inode.size
            inode.size = file_offset

        buffer_offset += to_write
        total_left - = to_write

    file_table[fd].current_offset=file_offset
    return buffer_offset
```