Petit voyage au centre d'un

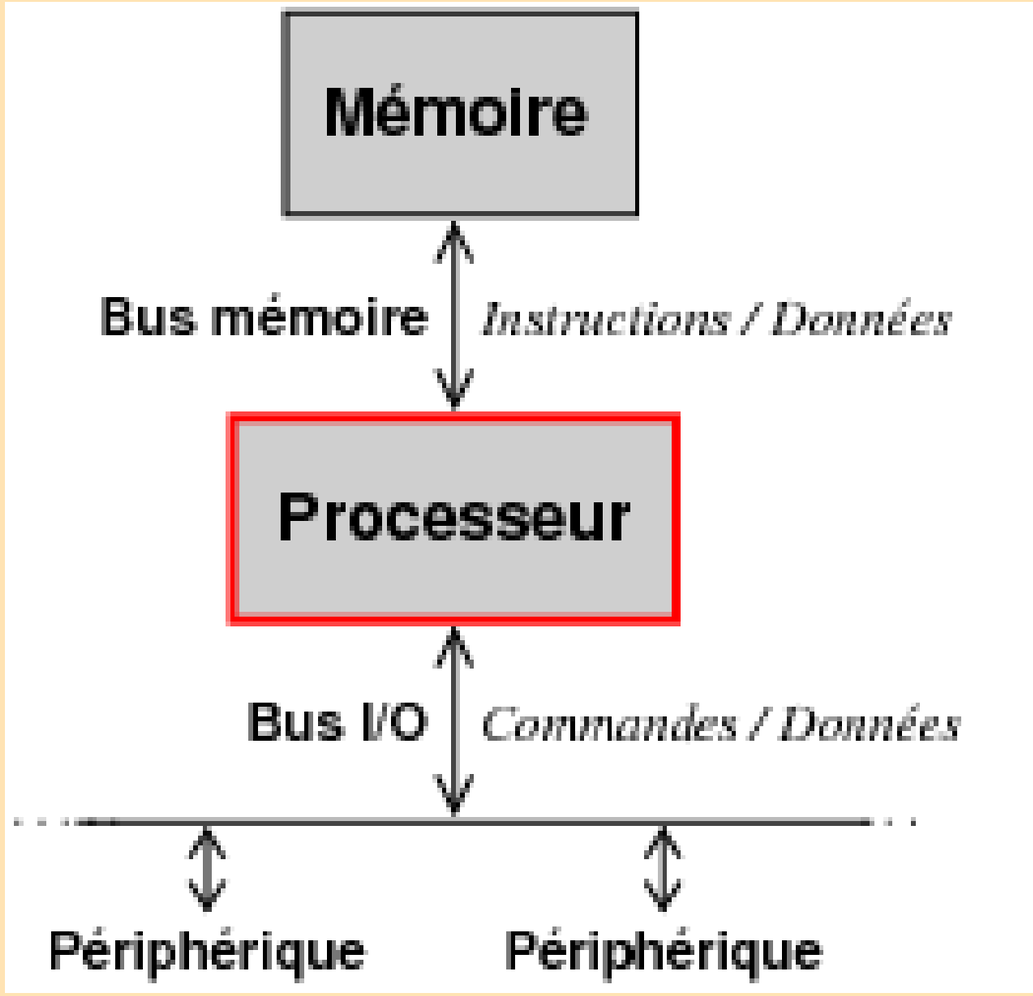
Système d'exploitation

Mercredi 5 et Jeudi 6 Juillet 2006

Renaud Lottiaux - renaud.lottiaux@kerlabs.com
Thomas Petazzoni - thomas.petazzoni@enix.org

Ordinateur ?

- Ordinateur
  - processeur
  - mémoire
  - périphériques
- Processeur : machine assez bête exécutant séquentiellement des instructions



Processeur

- Exécute séquentiellement des instructions, stockées en mémoire
- Dispose de plusieurs *registres* :
  - pointeur d'instruction *eip*
  - pointeur de pile *esp*
  - registres de travail (*eax, ebx, ecx, edx..)*
- Compilateur génère du code utilisant ces instructions et ces registres
- Se base sur des conventions d'appel pour les appels de fonctions / sous-fonctions

Code

```
#include <stdio.h>
int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

Compilateur

```
08048354 <main>:
8d 4c 24 04                lea    0x4(%esp),%ecx
83 e4 f0                   and    $0xfffffff0,%esp
ff 71 fc                   pushl  0xfffffffc(%ecx)
55                         push   %ebp
89 e5                      mov    %esp,%ebp
51                         push   %ecx
83 ec 04                   sub    $0x4,%esp
c7 04 24 b8 84 04 08       movl   $0x80484b8,(%esp)
e8 0b ff ff ff             call   804827c <puts@plt>
b8 00 00 00 00             mov    $0x0,%eax
83 c4 04                   add    $0x4,%esp
59                         pop    %ecx
5d                         pop    %ebp
8d 61 fc                   lea    0xfffffffc(%ecx),%esp
c3                         ret
```

Conventions d'appel

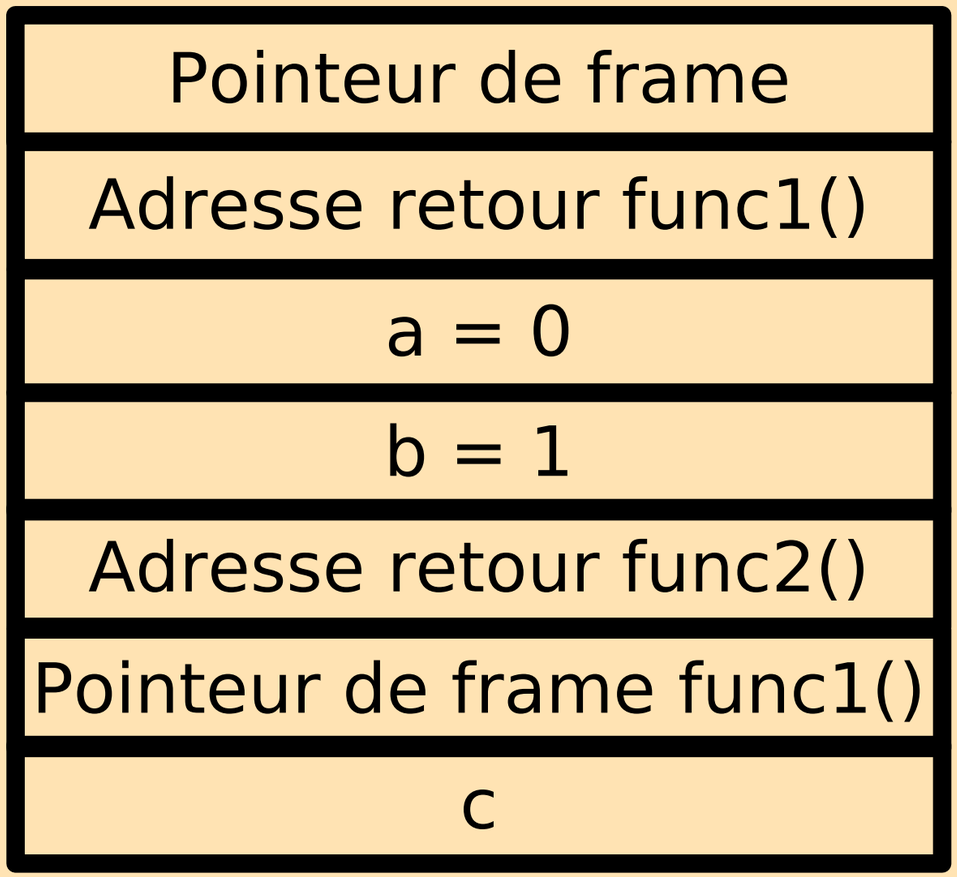
Avec **gcc** :

- Passage des paramètres sur la pile
- Allocation des variables locales sur la pile
- Utilisation de **ebp** comme pointeur de frame
- Valeur de retour dans **eax**

Conventions d'appel

```
int func2(int a, int b)
{
 int c;
 c = a + b;
 return c;
}
void func1(void) {
   func2(0, 1);
 }
```

```
00000014 <func1>:
   push   %ebp
   mov    %esp,%ebp
   sub    $0x8,%esp
   movl   $0x1,0x4(%esp)
   movl   $0x0,(%esp)
   call   2a <func1+0x16>
   leave
   ret
```

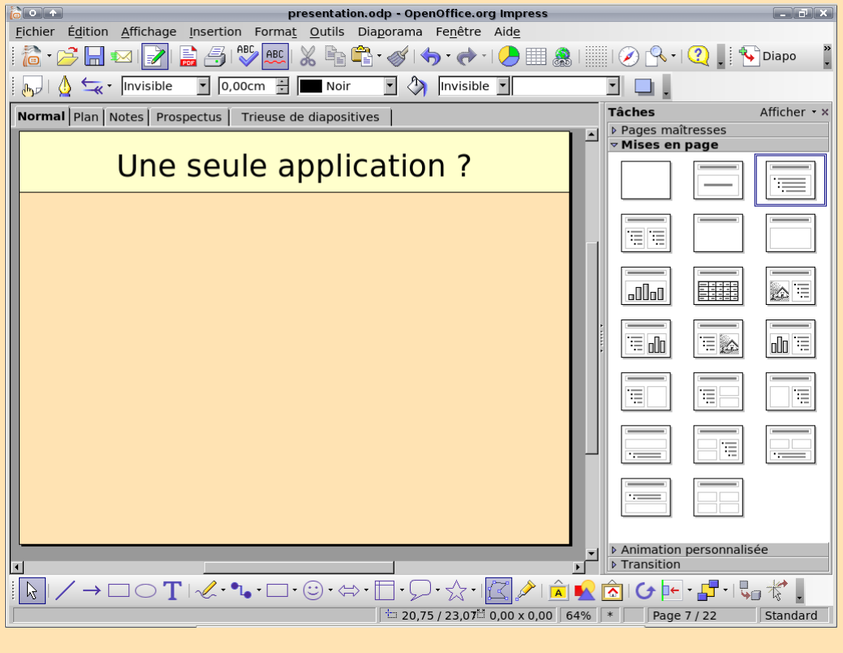


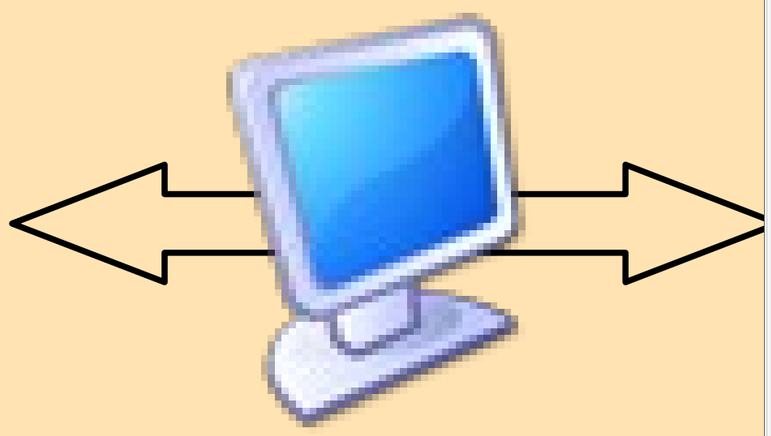

```
00000000 <func2>:
   push   %ebp
   mov    %esp,%ebp
   sub    $0x10,%esp
   mov    0xc(%ebp),%eax
   add    0x8(%ebp),%eax
   mov    %eax,0xfffffffc(%ebp)
   mov    0xfffffffc(%ebp),%eax
   leave
   ret
```

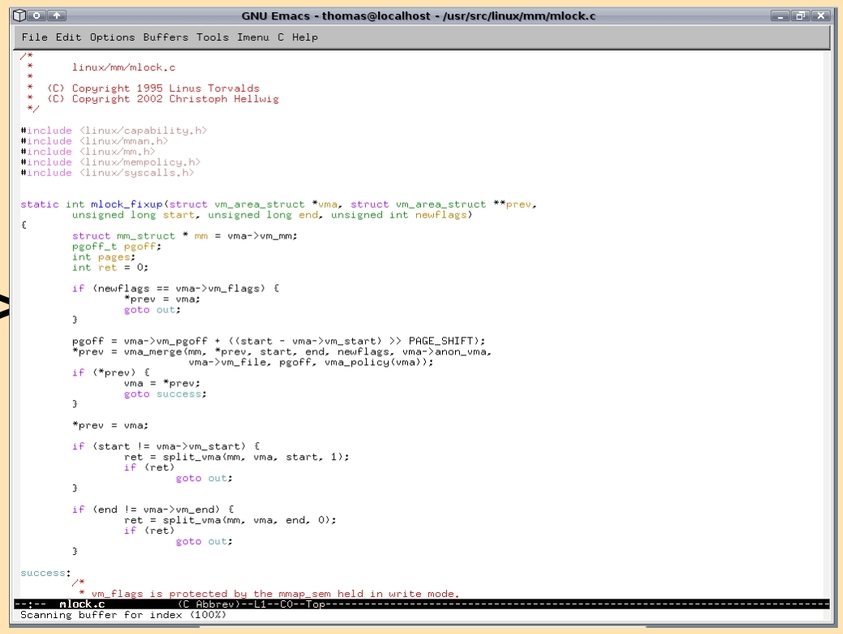
Interruption

- L'exécution séquentielle du code peut être interrompue par des **interruptions**
  - provenant du processeur directement, pour signaler une erreur : **exception**
  - provenant du matériel, pour signaler l'arrivée d'un événement : **IRQ**
  - provenant du logiciel
- Lorsqu'une interruption arrive, le processeur stoppe son exécution, et exécute un *gestionnaire d'interruption*

Une seule application ?







- On souhaite exécuter plusieurs applications
- Un ordinateur n'a qu'un seul processeur, une seule mémoire, un seul disque dur
- Besoin de *partager* les ressources

Rôle d'un OS

- Gérer et partager les ressources
  - Mémoire
  - Processeur
  - Périphériques
- Les rendre accessibles aux applications
- Deux types d'OS
  - à noyau monolithique
    - un seul bloc réalise toute la gestion et le partage des ressources, pour le compte des applications utilisateur
  - à micro-noyau
    - un petit noyau réalisant le strict minimum, et des applications utilisateur spécialisé pour réaliser le partage des ressource pour le compte d'autres applications

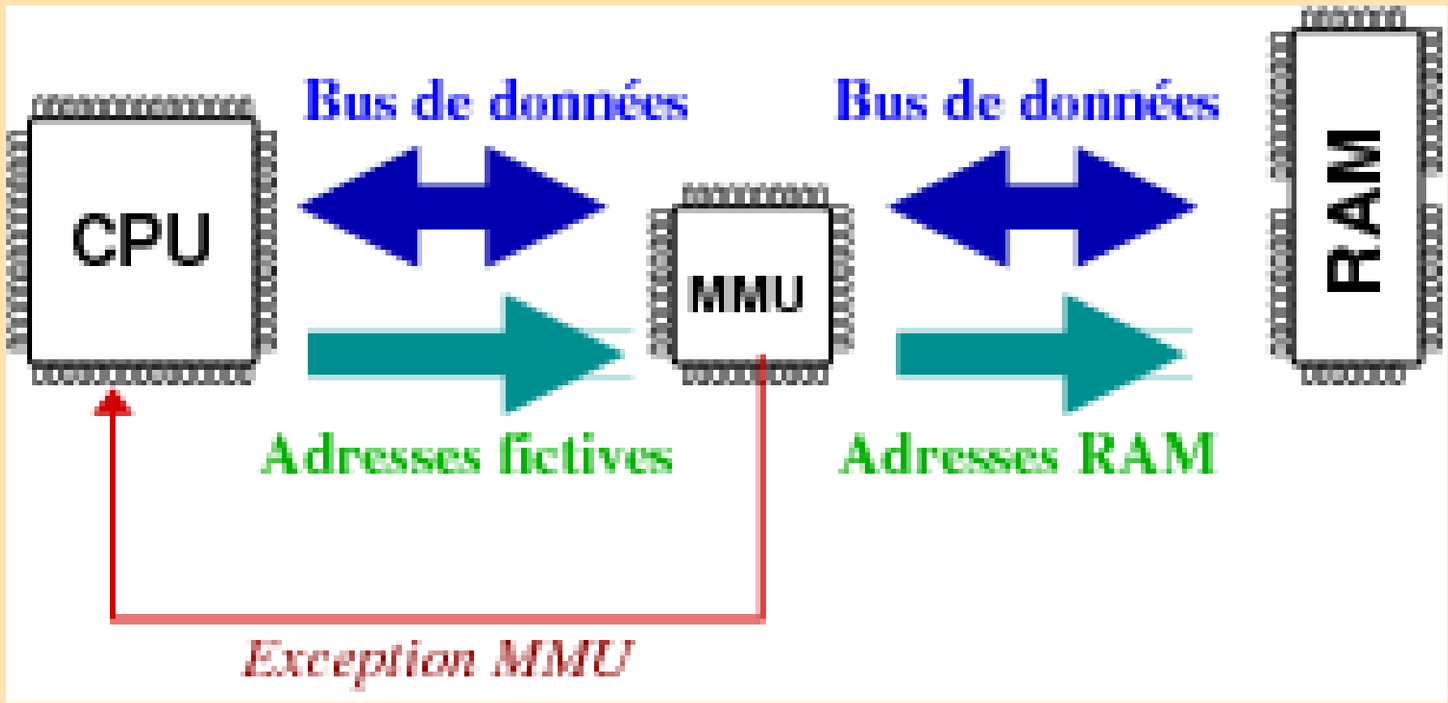
Mémoire physique

- Mémoire réellement disponible dans l'ordinateur – RAM
- L'OS doit gérer cette mémoire, i.e connaître les parties utilisées et non-utilisées
- Un seul espace d'adressage: pas de séparation entre les applications

Mémoire virtuelle

- Pour faire du multitâche robuste, deux types d'adresses :
  - adresses *physiques*
  - adresses *virtuelles*
- Conversion virtuelle -> physique réalisée par la MMU

- « Virtualisation » de la mémoire physique



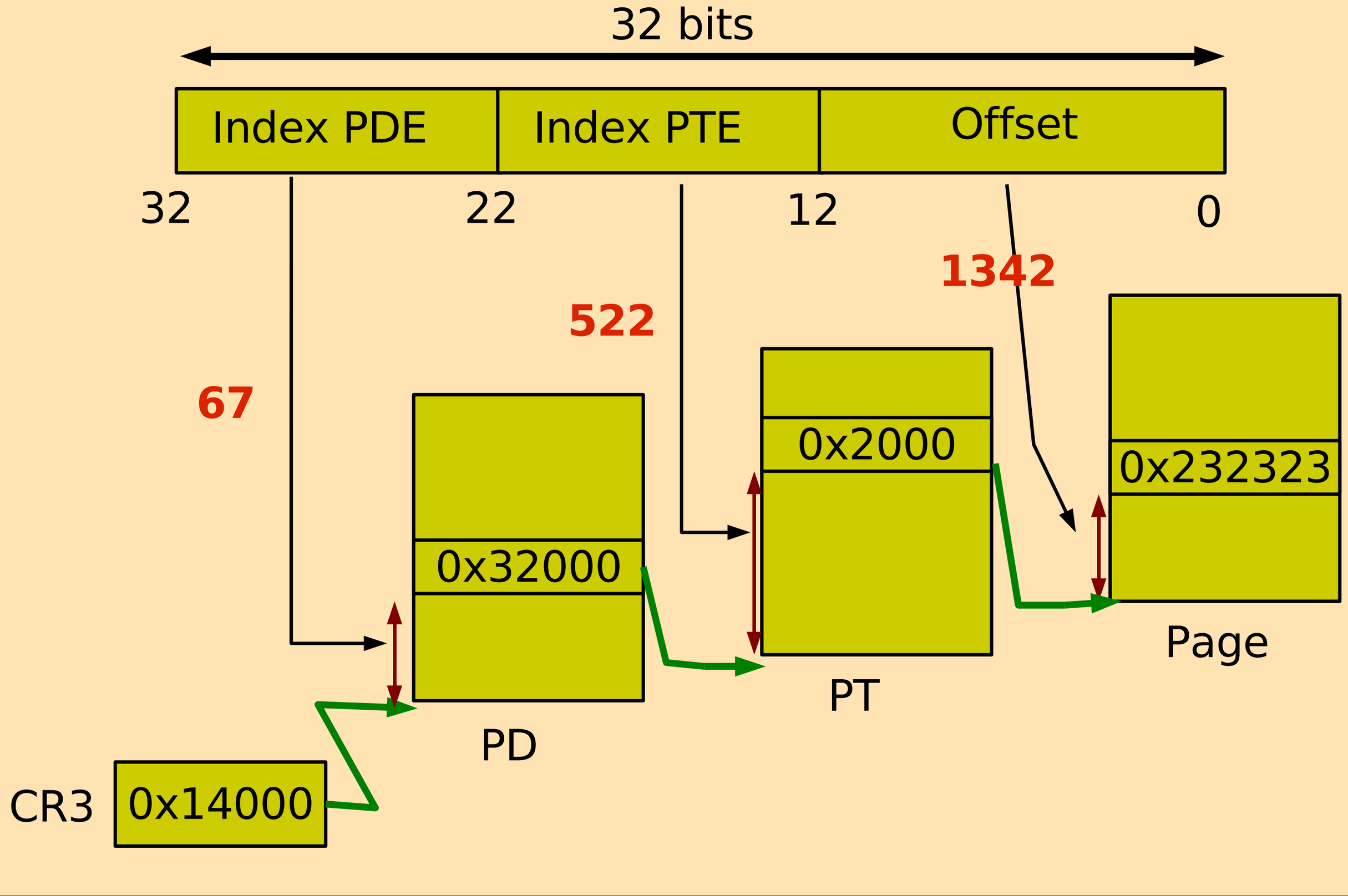
MMU

- La conversion virtuelle => physique
  - s'effectue selon des tables de traduction, construites par l'OS
  - est réalisée par la MMU (dans le processeur)
- Mécanisme de pagination
  - la mémoire virtuelle est découpée en *pages*
    - *4 Ko par défaut sur x86*
  - chaque page virtuelle est associée (ou non) à une page physique

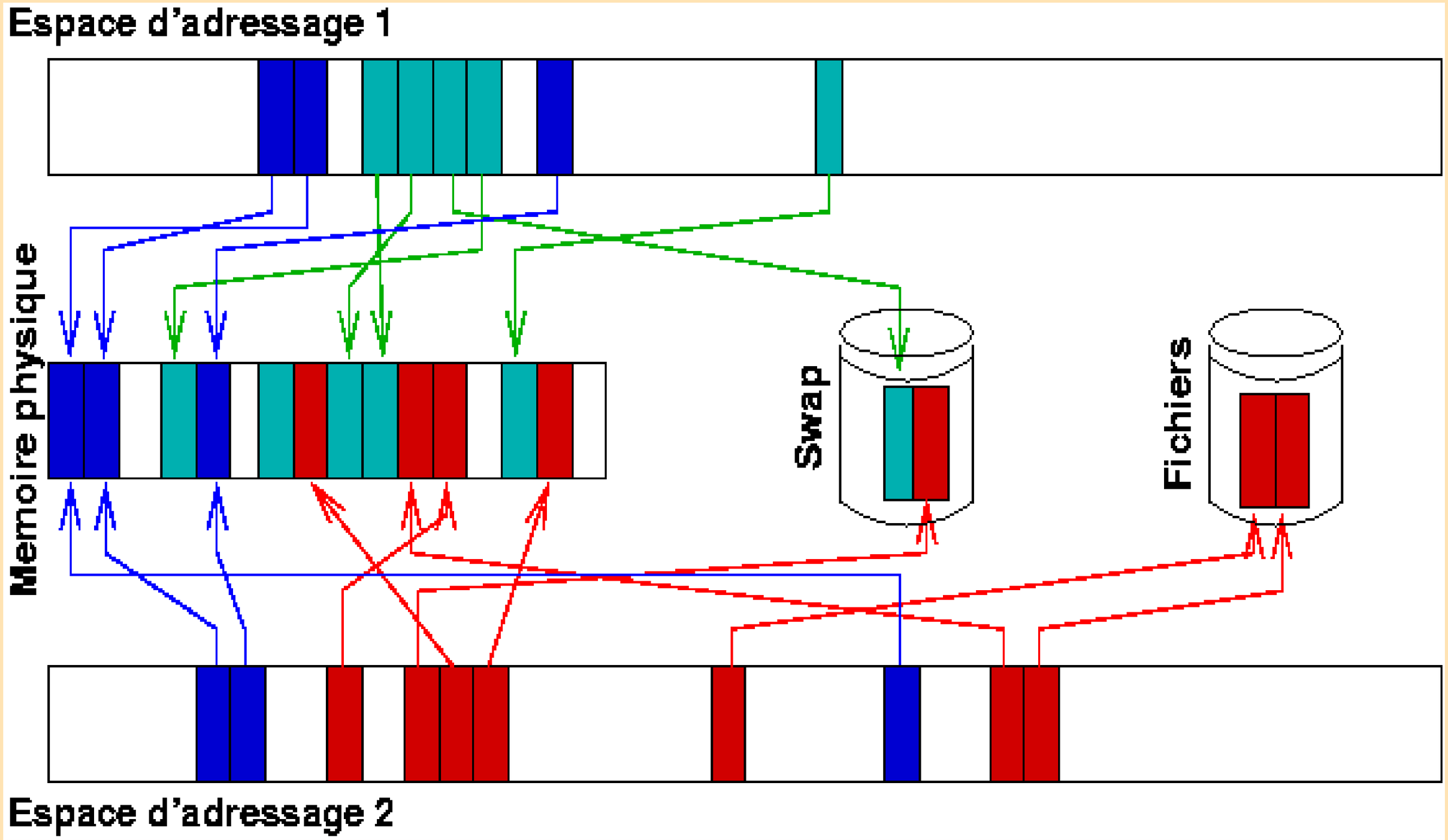
Pagination ?

- Sur x86, tables de traduction à 2 niveaux
  - répertoire de pages
  - table de pages
- Pour chaque espace d'adressage, **un unique** répertoire de pages (registre CPU)
- Un répertoire de pages pointe sur jusqu'à 1024 tables de pages
- Une table de pages pointe sur jusqu'à 1024 pages physiques
- 1024 tbl/rep * 1024 pg/table * 4 Ko = 4 Go
- Espace d'adressage de 4 Go = 2 ^ 32

Pagination


32 bits
Index PDE
Index PTE
Offset
32
22
12
0
1342
522
67
0x2000
0x232323
0x32000
Page
PT
PD
CR3
0x14000

Espace d'adressage

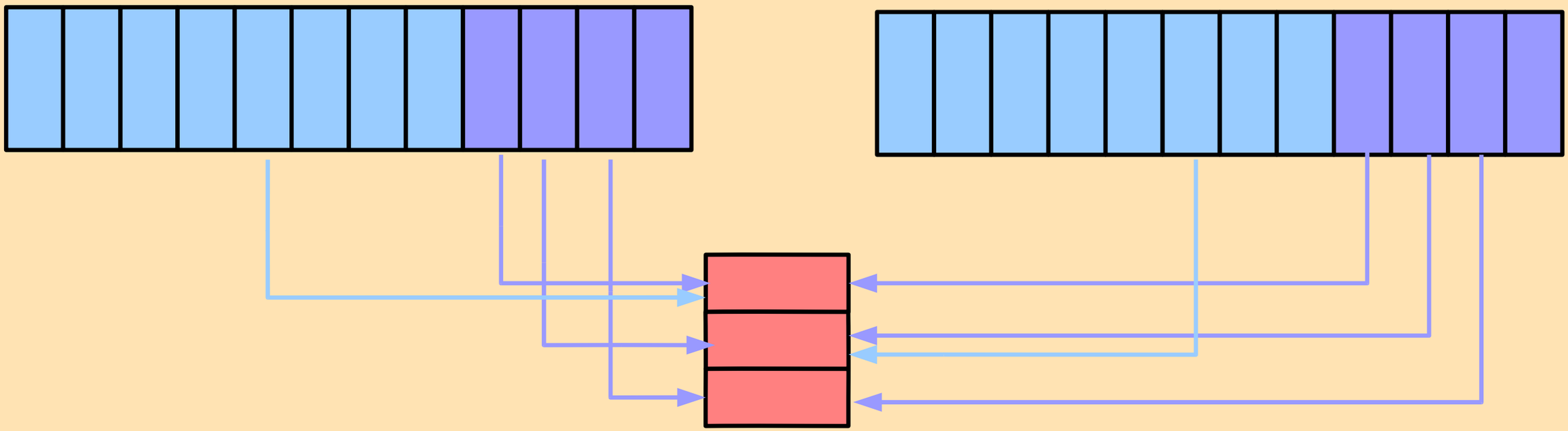


Et donc ?

- Espaces d'adressage
  - cloisonnement
  - adresses fixes
- Partage de pages
- Chargement à la demande
- Taille mémoire virtuelle supérieure à la taille mémoire physiquement disponible
  - swap

Découpage espace adressage

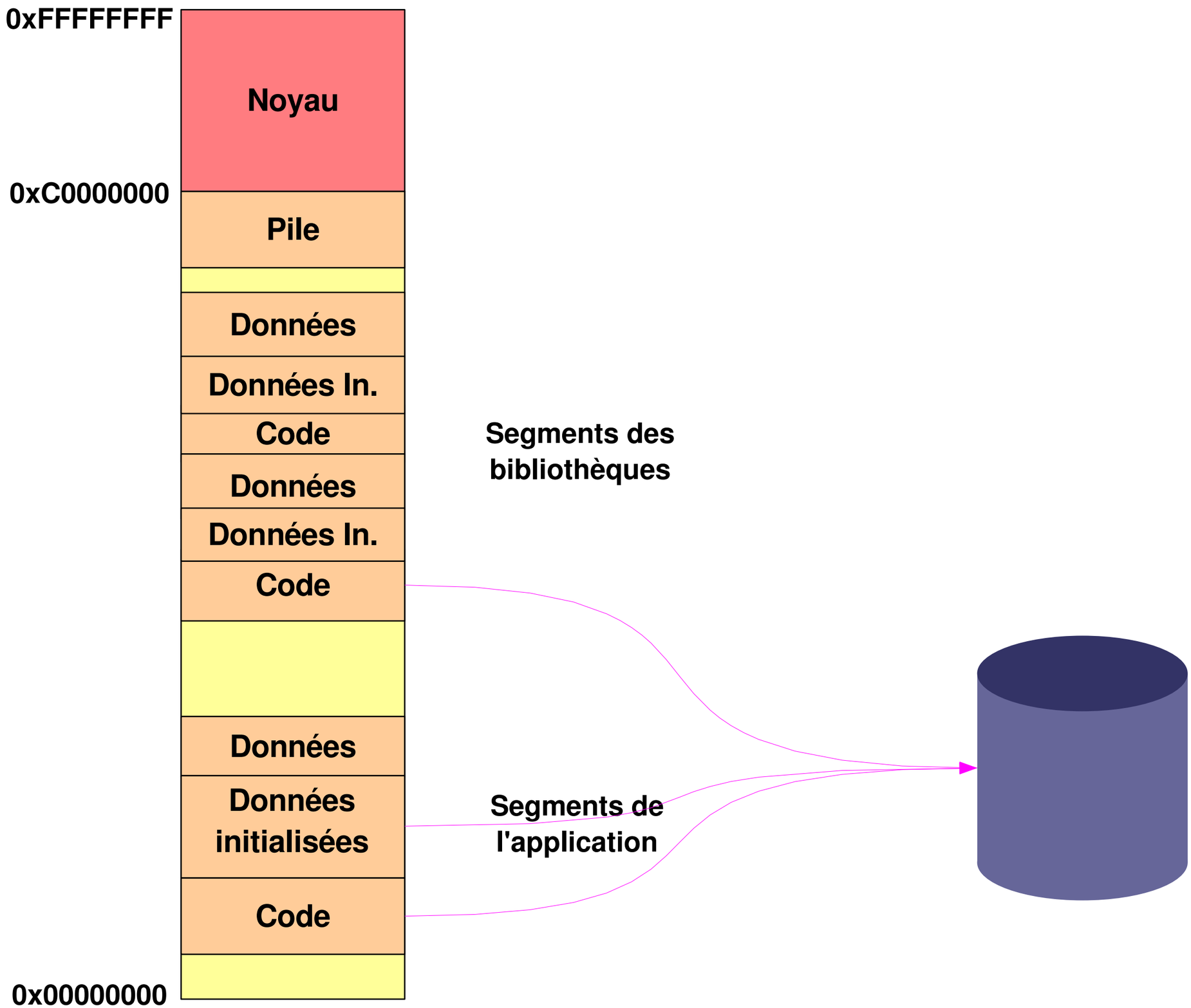
- Sous Linux, découpage de tous les espaces d'adressage en deux parties
  - Une partie noyau (3G-4G)
    - identique dans tous les espaces d'adressage
    - identity-mapping de la mémoire physique
  - Une partie utilisateur (0G-3G)
    - spécifique à l'application en cours d'exécution



Découpage partie utilisateur

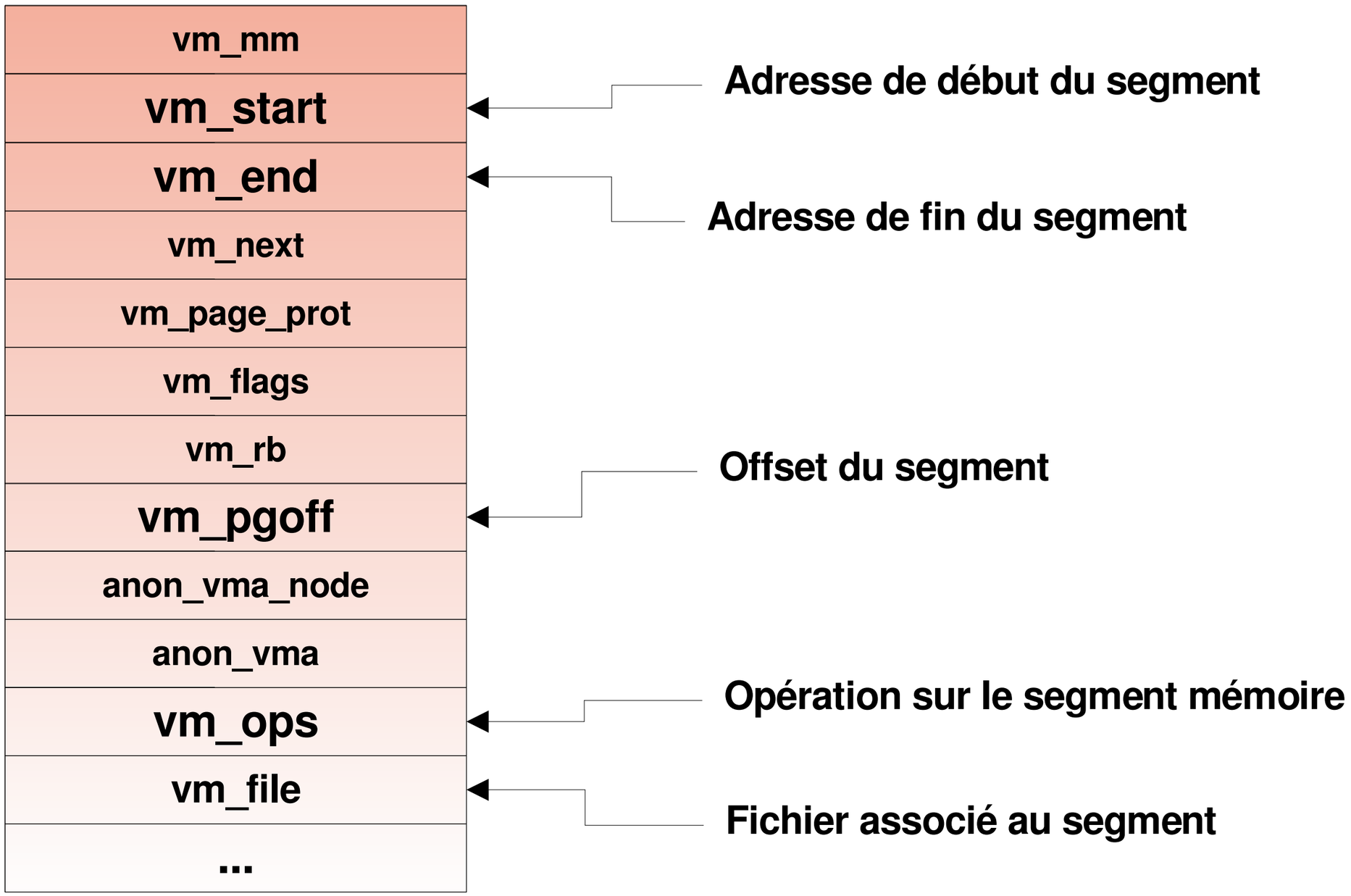
- En *régions virtuelles*
- Une région virtuelle pour chaque partie de l'application
  - Code du programme
  - Données du programme
  - Code des bibliothèques partagées
  - Données des bibliothèques partagées
  - Tas
  - Pile

Segments mémoire



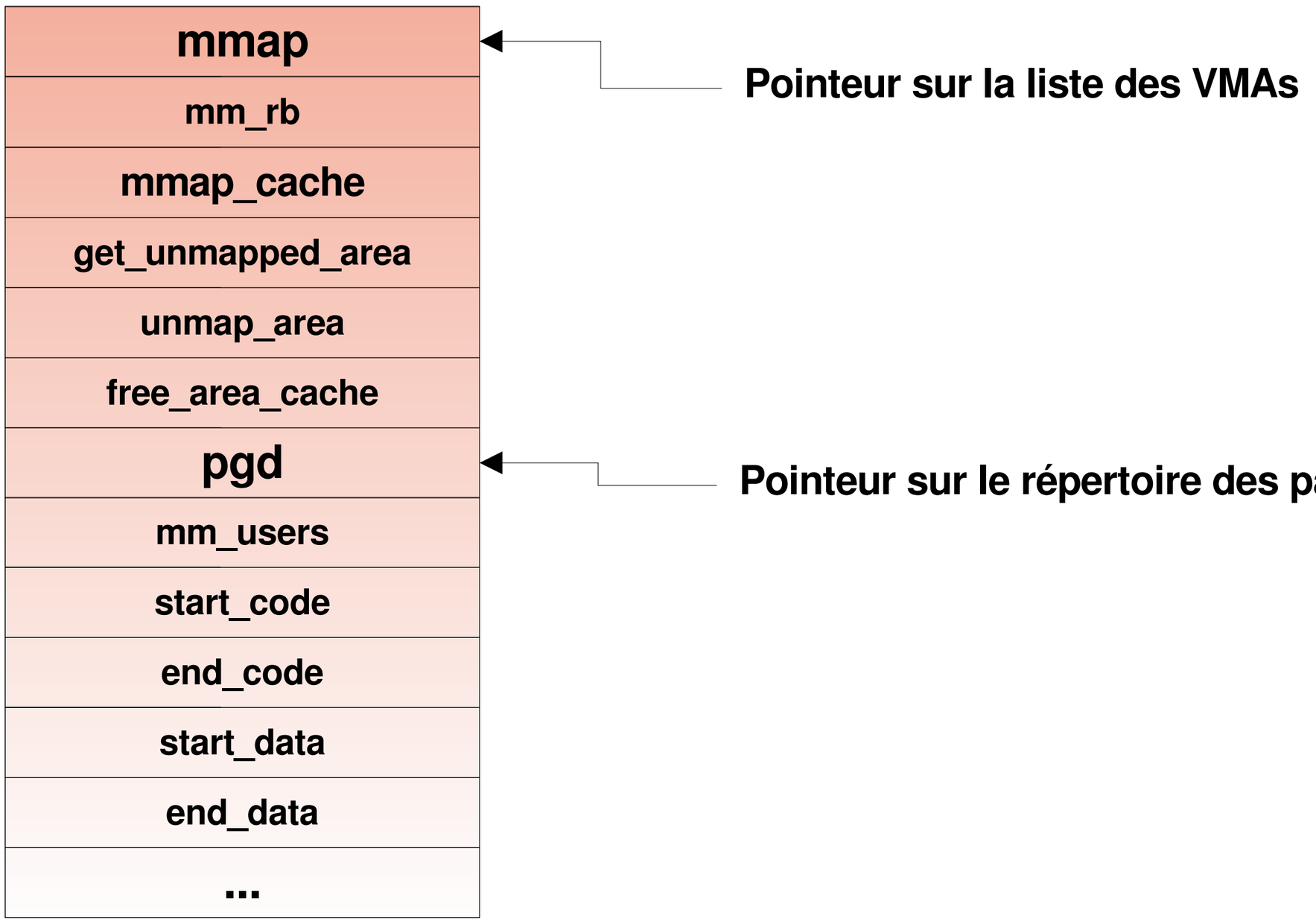
vm_area_struct (Virtual Memory Area)

- Stocke les informations d'un segment mémoire
- Une structure par segment mémoire

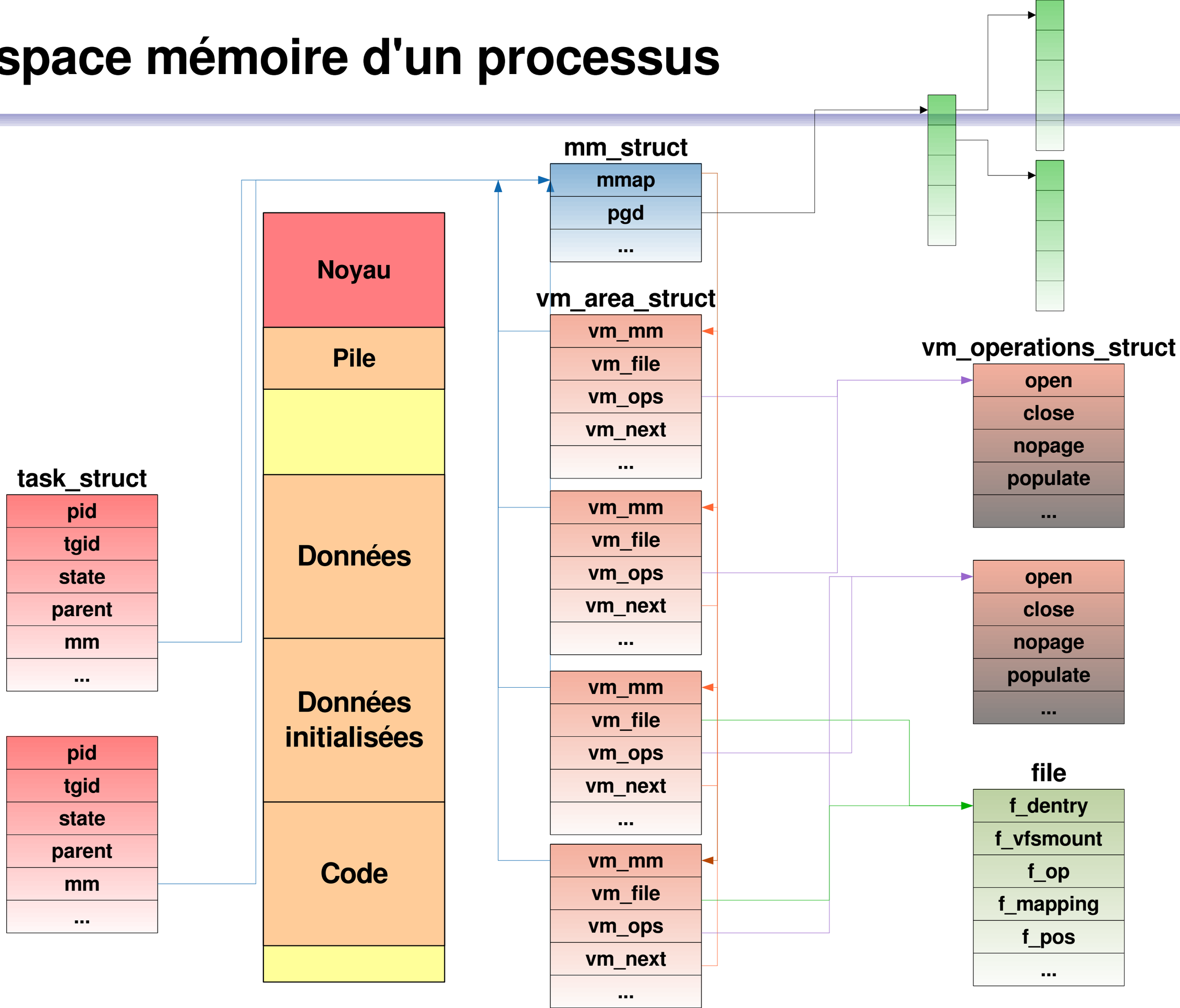


Descripteur d'espace d'adressage : mm_struct

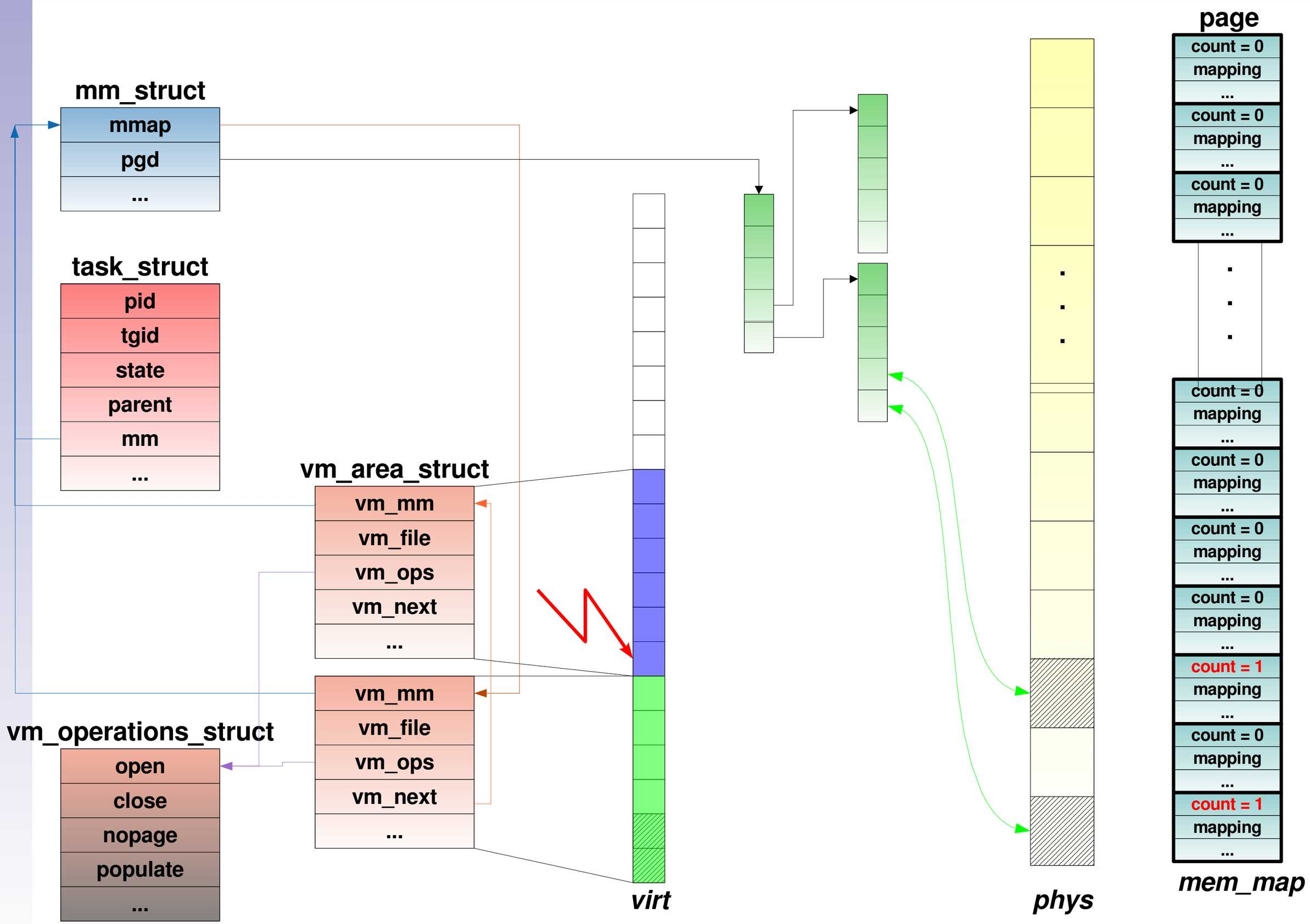
- Stocke les informations sur l'espace d'adressage d'un process
- Une structure par processus



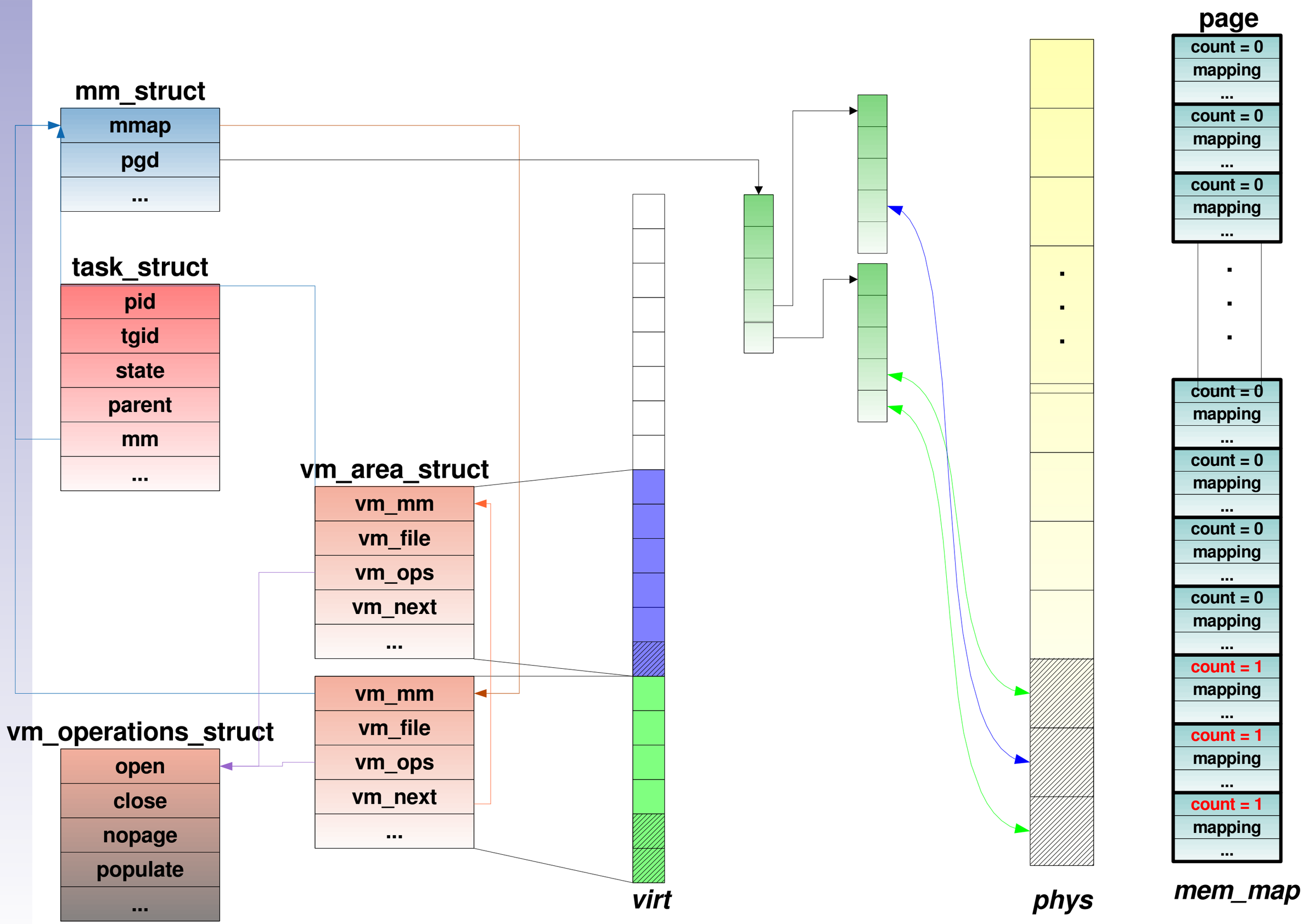
Espace mémoire d'un processus



Allocation paresseuse

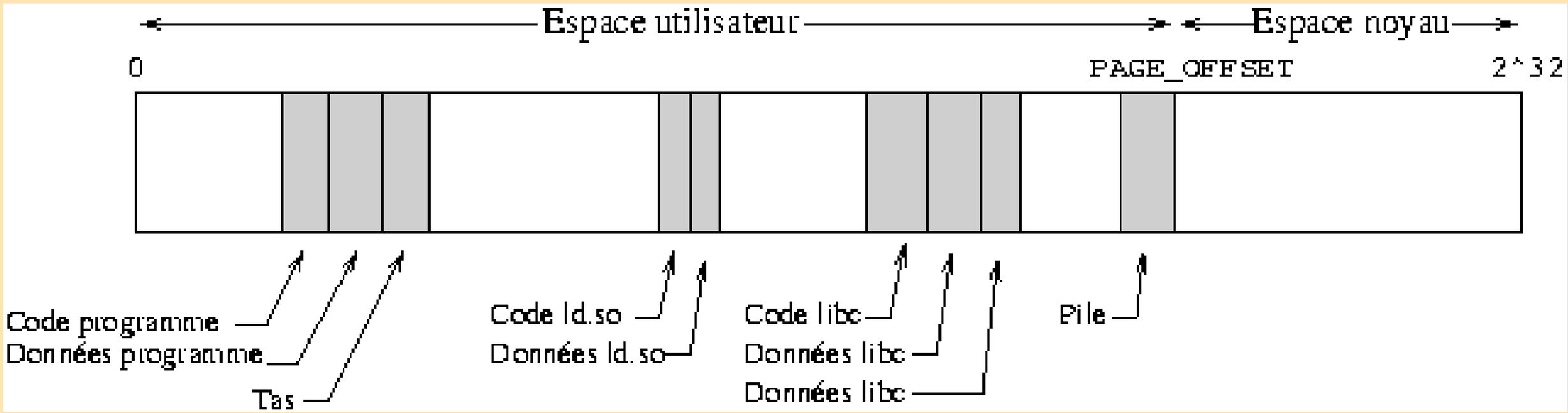


Allocation paresseuse



Régions virtuelles

$ cat /proc/[pid]/maps



Région virtuelle

- Une région virtuelle correspond
  - soit à un fichier, *file mapping*
  - soit à de la mémoire allouée dynamiquement, *anonymous mapping*

- Régions créées
  - lors du lancement de l'application
  - grâce aux *appels systèmes* de la famille mmap()

- Démo mmap() + *cat /proc/[pid]/smaps*

Interlude: appel système

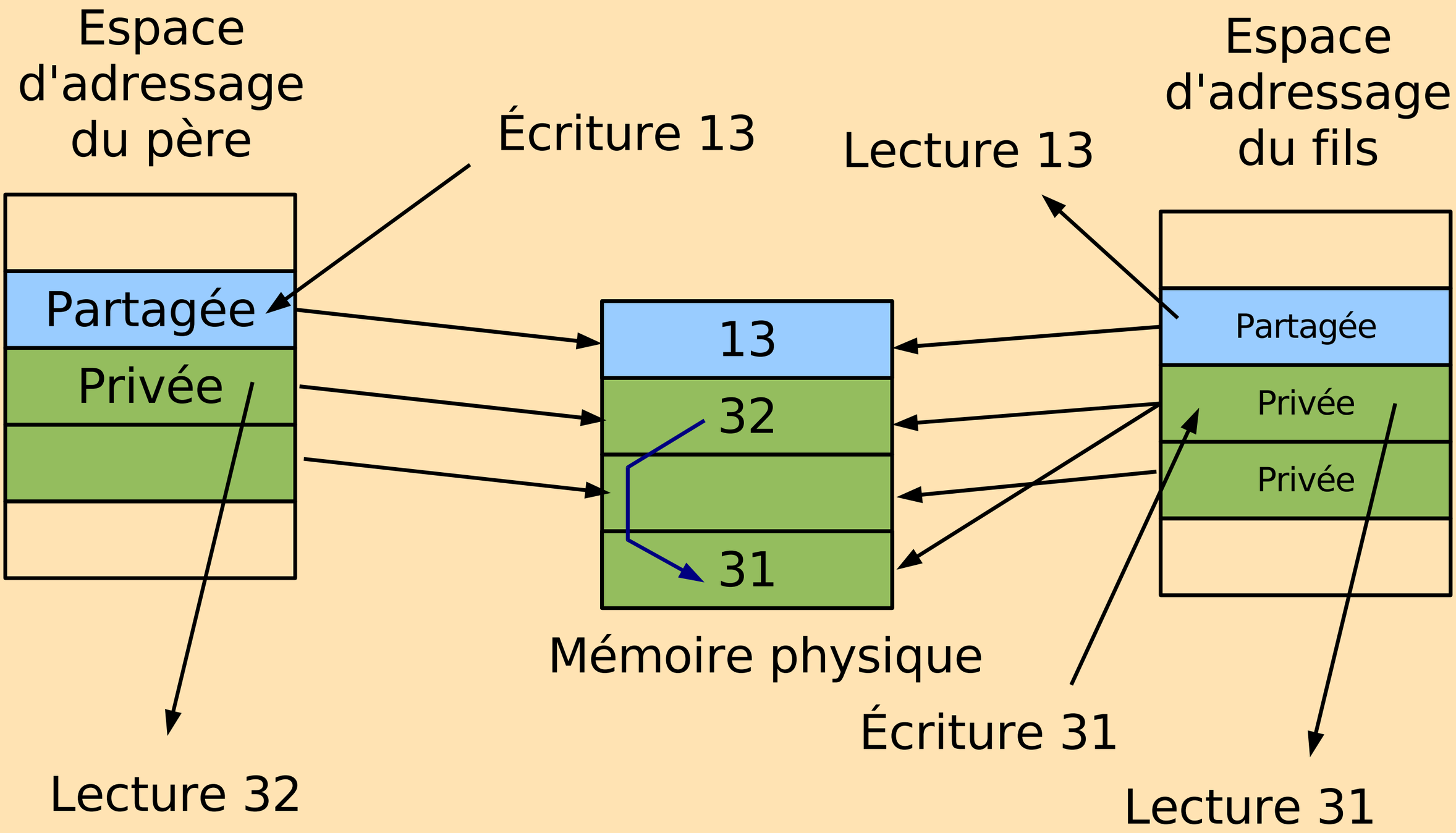
- Parmi les buts de l'OS: fournir des services aux applications
- Le noyau s'exécute en mode privilégié, les applications utilisateurs dans un mode non privilégié
- Un simple appel de fonction ne peut pas marcher
- Mécanisme contrôlé de passage en mode noyau
  - appel système
  - basé sur une interruption logicielle

Démo !

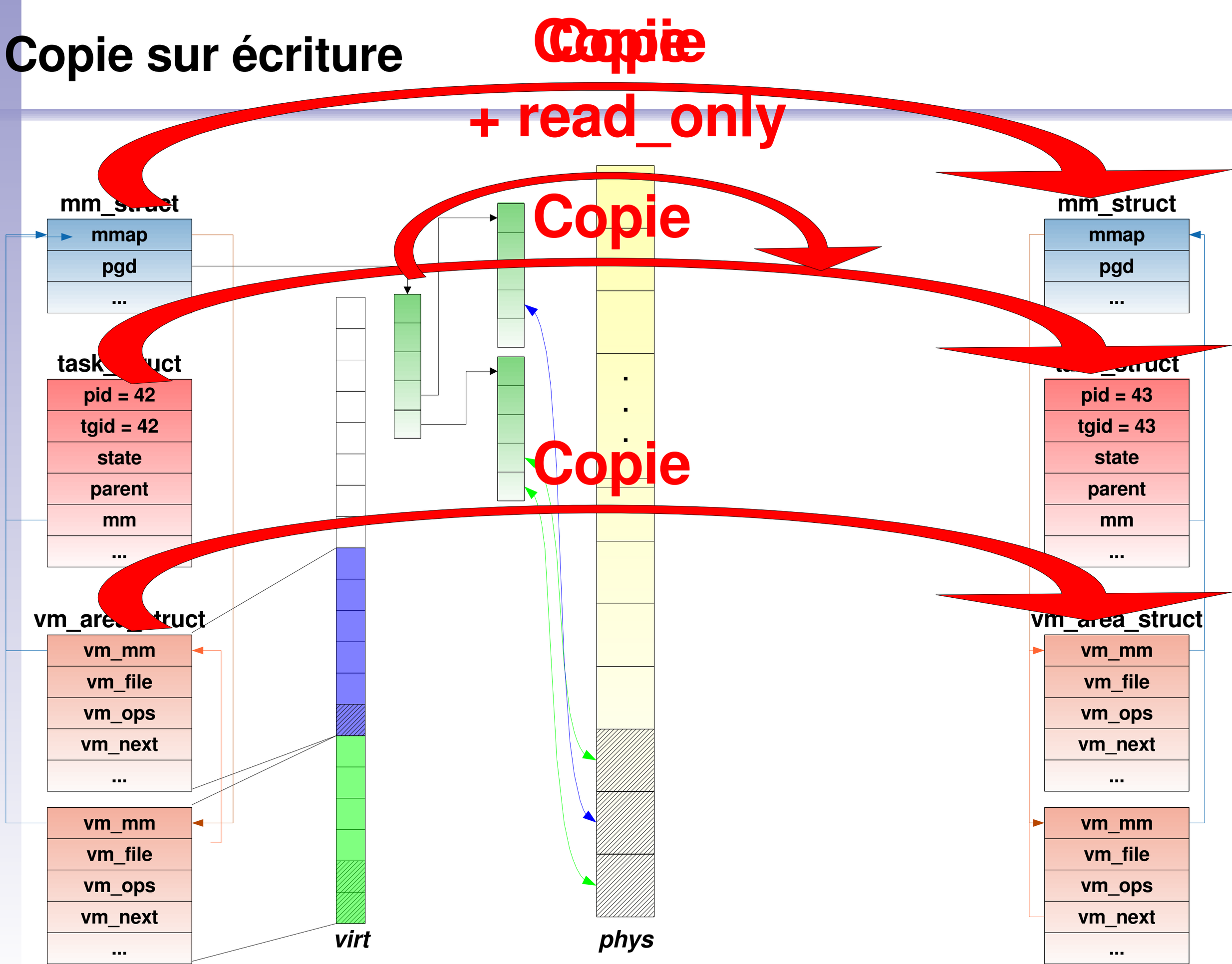
fork(), ainsi naquit le processus

- Création de nouveau processus par l'appel système fork()
- Dans un nouvel espace d'adressage
- Copie du processus parent
  - duplication des régions virtuelles
  - mais pas de leur contenu !
- Deux types de régions:
  - privée : modifications non partagées entre fils
  - partagée : modifications visibles entre fils
- Mécanisme de Copy-on-Write pour la différenciation

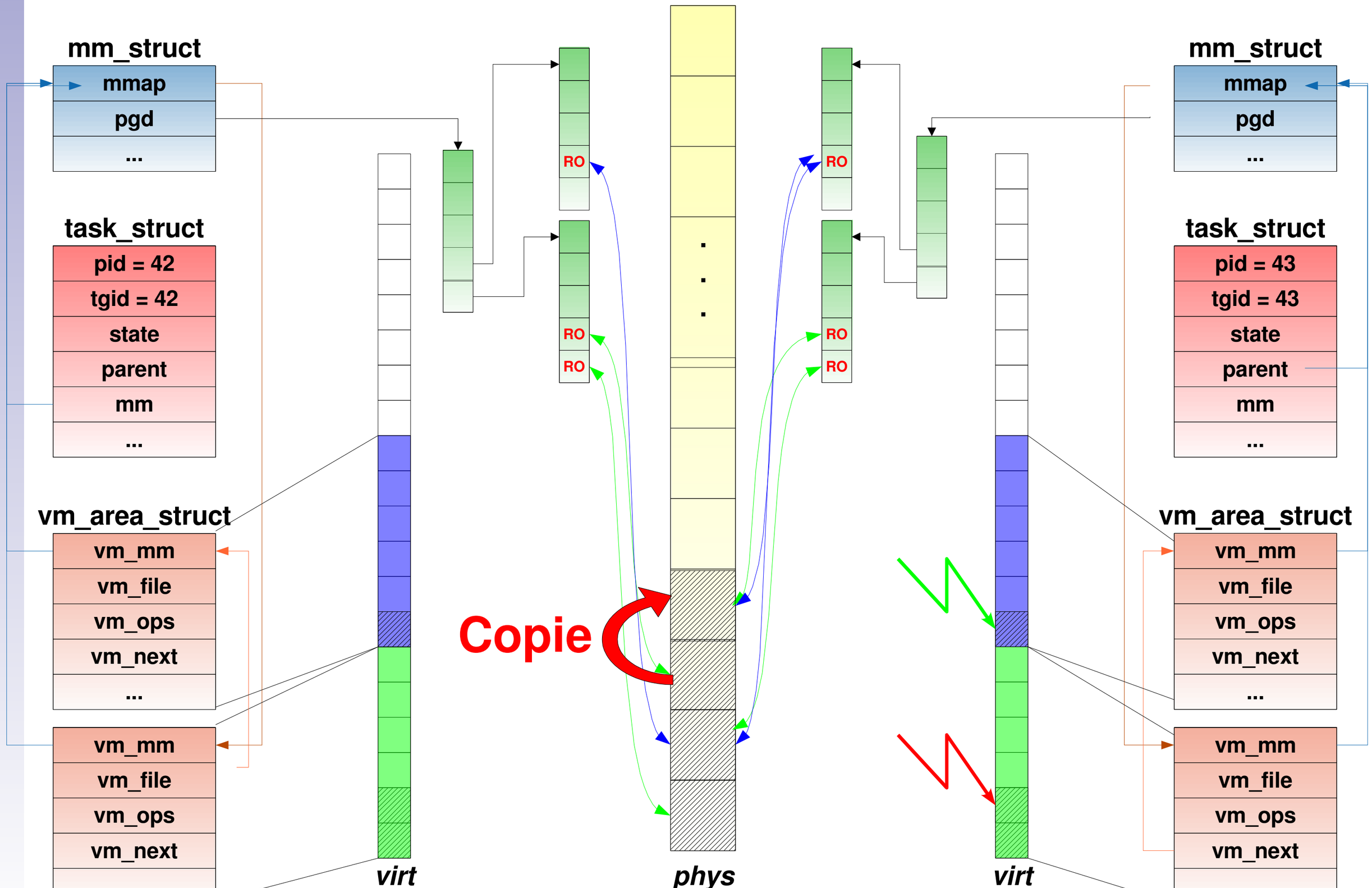
fork() et Copy-on-write



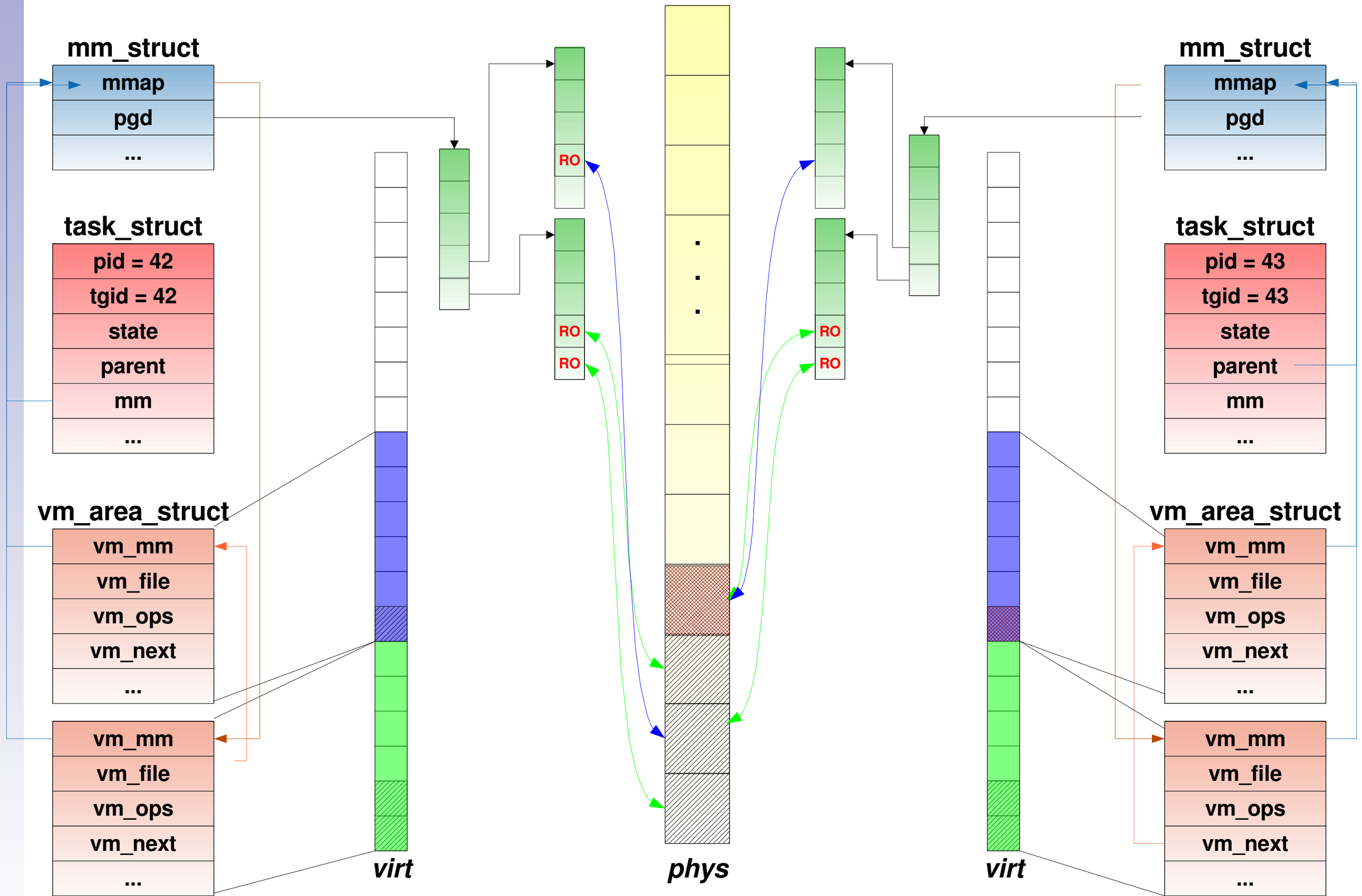
Copie sur écriture



Copie sur écriture



Copie sur écriture



Gestion du défaut de page

- Résumons la gestion d'un défaut de page

- Accès à une adresse hors région: SIGSEGV
- Page en lecture seule accédée en écriture
  - Région en lecture/écriture -> Copy-on-Write
  - Sinon -> SIGSEGV
- Sinon
  - Chargement à la demande
    - Anonyme: allocation d'une page initialisée à 0
    - Fichier: allocation d'une page initialisée au contenu d'une portion du fichier

exec(), exécutons autre chose

- fork() duplique le parent, donc créé un processus qui exécute le même code
- Besoin d'exécuter d'autres choses: appel système exec()
- Remplace les régions virtuelles du processus courant par l'image d'un autre programme
- Mappe le binaire et le chargeur de bibliothèques
- Exécute le chargeur de bibliothèques qui mappe les bibliothèques

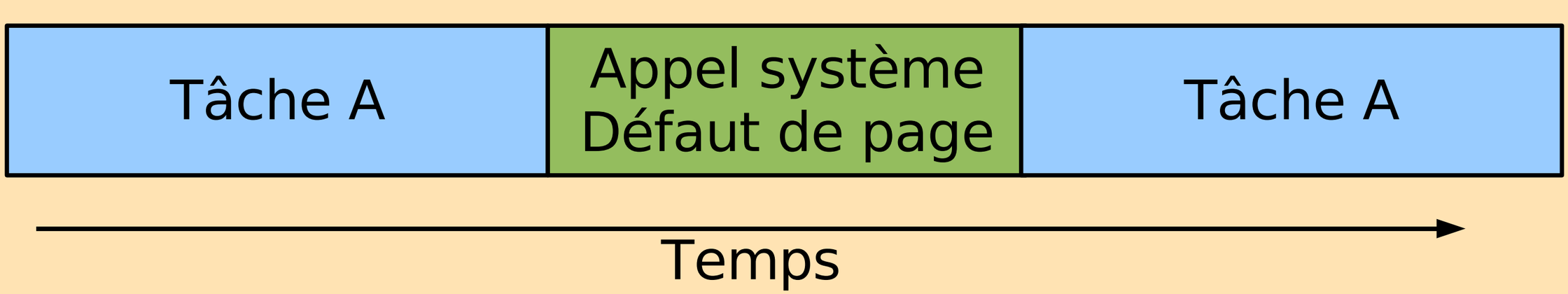
Démo !

Mémoire, au niveau utilisateur

- Utilisation de malloc() et free(), fonctions de la bibliothèque standard C
- Utilise les appels systèmes
  - brk() pour agrandir la taille du tas
  - mmap() pour allouer des grandes zones de mémoire anonyme

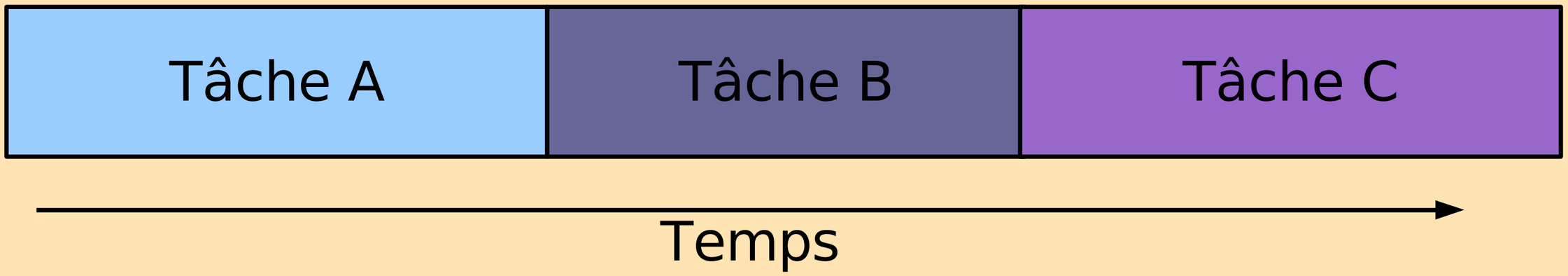
Démo !

Partage du processeur



Et mes autres applications,
elles s'exécutent quand ?

Partage du processeur



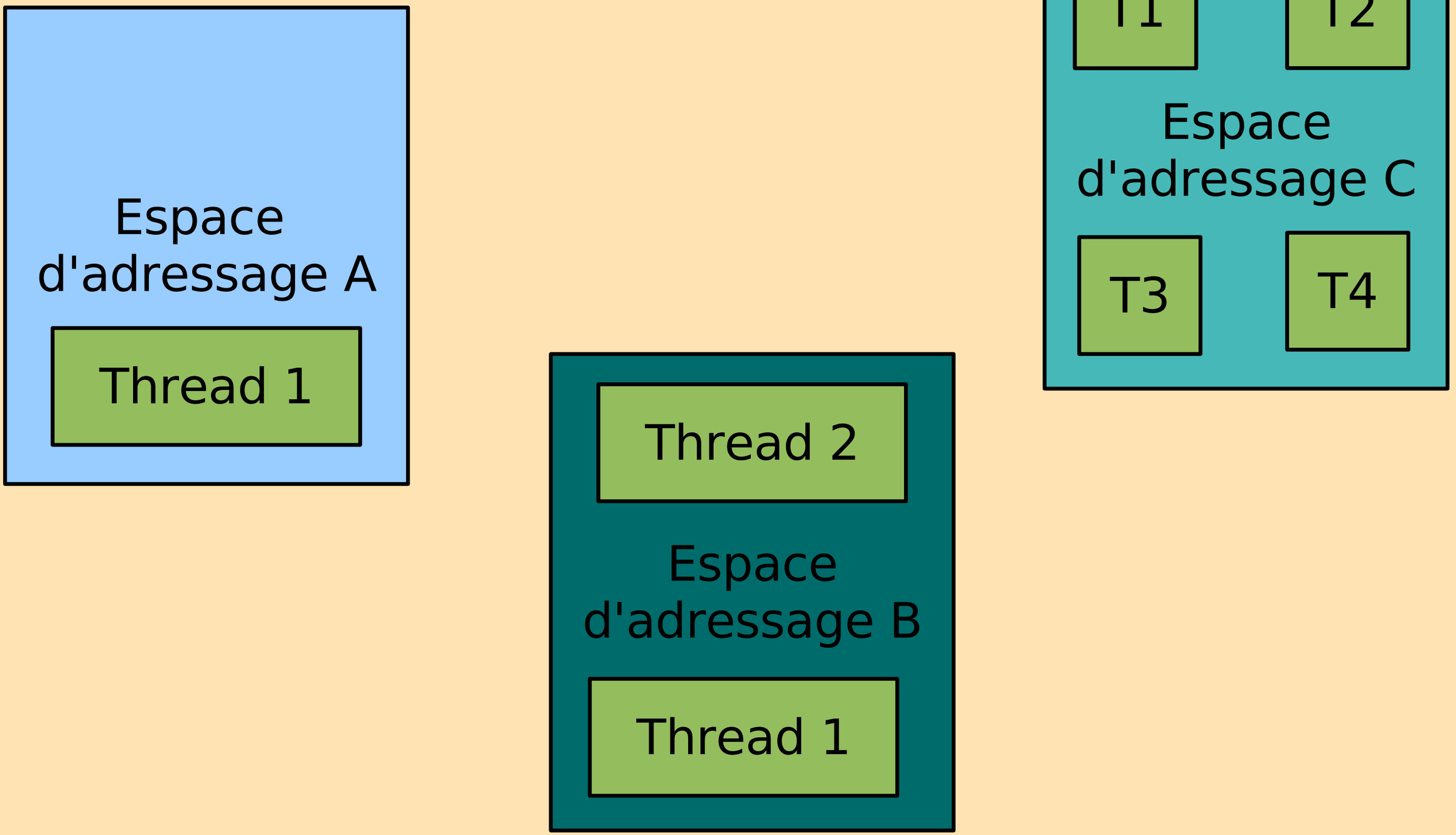
- Multitâche
  - illusion d'exécution en parallèle de plusieurs flots d'instruction
- Chaque flot d'instruction est appelé *thread*
- Le passage d'un thread à un autre implique un *changement de contexte*

Thread

- Chaque processus dispose
  - d'un espace d'adressage
  - d'un thread – flot d'exécution
- On peut créer d'autres threads dans l'espace d'adressage courant
  - bibliothèque Pthread, utilise l'appel système clone()
- Tous les threads d'un même espace d'adressage accèdent aux mêmes données

```
5603 ?           -      15:07 soffice.bin -impress
   - -           Sl     15:07 -
   - -           Sl      0:00 -
   - -           Sl      0:00 -
   - -           Sl      0:00 -
```

Thread


Espace
d'adressage A
Thread 1
T1
T2
Espace
d'adressage C
T3
T4
Thread 2
Espace
d'adressage B
Thread 1

Thread

- Deux types
  - noyau: exécuté avec tous les privilèges, dans n'importe quel adressage, pour effectuer des tâches spécifiques au noyau
  - utilisateur: exécution du flot d'instruction d'une application

```
 151 ?        S      0:00 [kswapd0]
 152 ?        S<     0:00 [aio/0]
5603 ?        -     15:07 soffice.bin -impress
   - -        Sl    15:07 -
   - -        Sl     0:00 -
   - -        Sl     0:00 -
   - -        Sl     0:00 -
```

Changement de contexte

- Changement de contexte T1 -> T2
  - sauvegarde du contexte de T1
  - restauration du contexte de T2
- Prise de photographie
- Contexte: ensemble des registres du processeur
- Sauvegardé dans la pile noyau du thread

Changement de contexte

CPU

eip = 443
eax = 24
ebx = 253
esp = 7676

Pile noyau T1

ptr

42
24
67
4242

ptr

Pile noyau T2

ptr

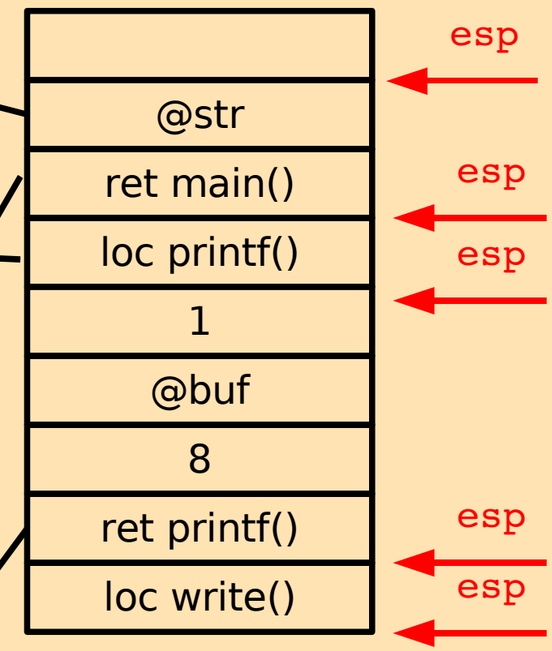
643
27
253
7676

ptr

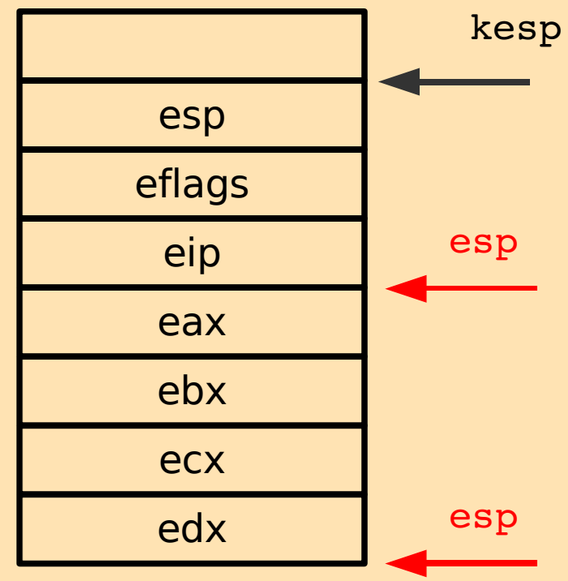
Changement de contexte

```
int main(void) {
  printf("Bonjour\n");

        int printf(const char *fmt, ...) {
         ...
        write (1, "Bonjour\n", 8);


            int write(int fd, const char *buf, size_t
            len) {

                movl $2, %eax
                ...

                int $0x80

                       SAVE_ALL

                       sys_write()

                       RESTORE_ALL

                 movl %eax, ret

               return ret;
            }

        }

  return 0;
}
```

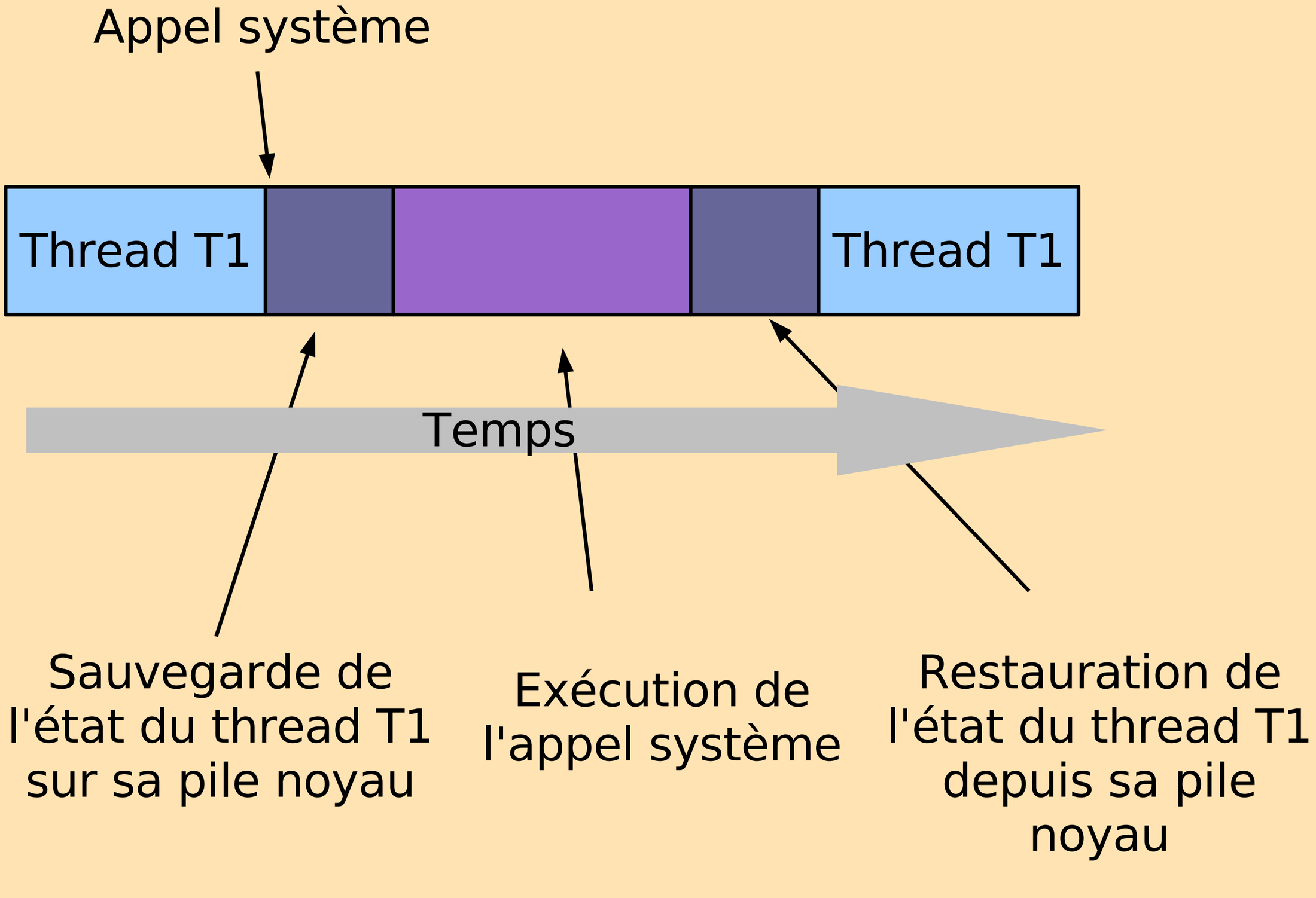
Pile utilisateur



Pile noyau

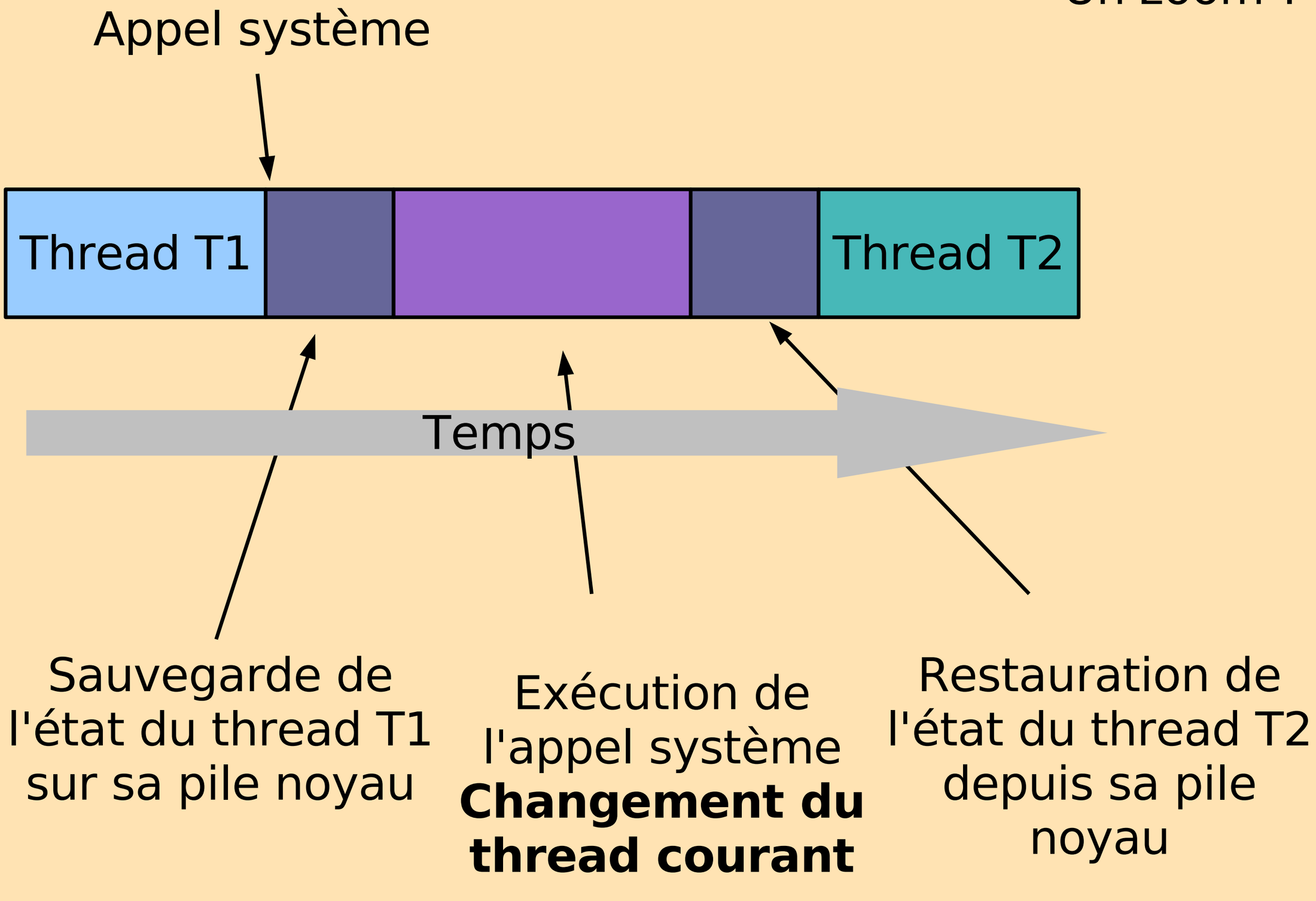


Changement de contexte



Changement de contexte

Un zoom ?



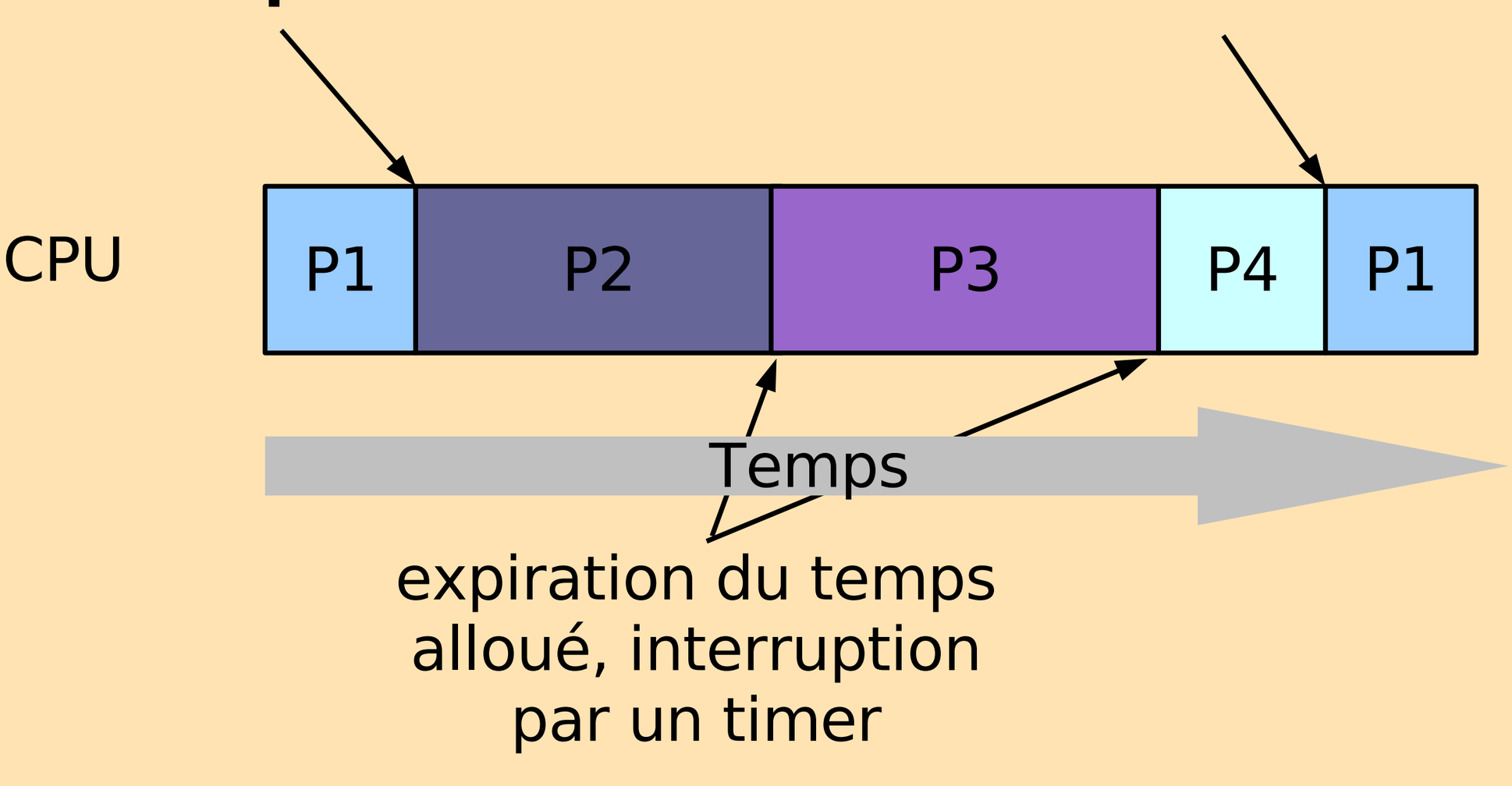
Ordonnancement

- Chaque thread a un état
  - prêt
  - en attente
  - en exécution
- Le processeur est passé à un autre thread quand :
  - le thread courant est bloqué (en attente d'I/O, ou d'un autre événement)
  - le thread courant a demandé à passer son tour (multitâche coopératif)
  - le thread courant a utilisé tout le temps qui lui a été allouée (multitâche **préemptif**)

Ordonnancement

read() sur l'entrée standard (clavier)
=> processus bloqué

Appui sur le clavier, P1 est débloqué et prêt à l'exécution



Ordonnanceur

- Partie du système qui choisit le prochain thread à exécuter en fonction de leurs états
- Ordonnanceur simple: round-robin
- En pratique, ordonnanceurs plus complexes
  - priorités
  - ordonnanceur en *O(1)*
  - temps-réel (dur, mou)

Synchronisation

- Besoin de synchronisation entre threads, pour
  - exclusion mutuelle : gestion de l'accès à des ressources partagées
  - rendez-vous temporels : signalisation d'un évènement

Exclusion mutuelle

Thread A et Thread B exécutent le même code dans le même espace d'adressage, ils partagent donc la ressource 'a'

Code

```
int a;
int f()
{
  a = 0;
  a++;
  return a;
}
```

f() = 1

Thread A

```
a = 0;
a++;
return a;
```

f() = 1

Thread B

```
a = 0;
a++;
return a;
```

f() = 1

Temps

Thread A

```
a = 0;
a++;

return a;
```

f() = 0

Thread B

```
a = 0;

a++;
return a;
```

f() = 1

Temps

Exclusion mutuelle

- Opérations atomiques
  - évitement des accès concurrents: impossible d'être interrompu
  - opérations sur les bits (test, set, change, clear) et sur les variables (set, sub, inc_and_test)

- a += 1 n'est pas atomique

```
mov     0x8049590,%eax
inc     %eax
mov     %eax,0x8049590
```

- Opération atomic_add() du noyau

```
atomic_add(int i, atomic_t *v) {
 __asm__ __volatile__(
          LOCK_PREFIX "addl %1,%0"
          :"=m" (v->counter)
          :"ir" (i), "m" (v->counter));
}
```

Exclusion mutuelle

- Spinlocks
  - désactivation des interruptions (en UP)
  - opération exécutée en un seul tenant
  - en SMP, attente active sur l'autre processeur
  - pas le droit de bloquer, réservé pour de courtes sections
- Mutex
  - pas d'attente active, les threads bloqués sont endormis
  - possibilité de bloquer

Exclusion mutuelle

Code

```
int a;
lock_t a_lock;
int f()
{
  int c;
  lock(a_lock);
  a = 0;
  a++;
  c = a;
  unlock(a_lock);
  return c;
}
```

Rendez-vous

- Utilisation d'un sémaphore
- Un compteur, deux opérations
  - down, compteur--, bloque quand compteur < 0
  - up, compteur++, réveille

```
semaphore_t keyboard_sem;
int read_keyboard()
{
  down(&keyboard_sem);
  c = kbd_buf[rd];
  rd++;
  return c;
}
```

```
int keyboard_int()
{
  kbd_buf[wrt] = car;
  wrt++;
  up(&keyboard_sem);
}
```

Synchro utilisateur

- Sémaphores
  - Même principe que pour les sémaphores noyau
  - semget(), semctl(), semop() (ipc/sem.c)
- Files de messages
  - Mécanisme de synchro et de communication
  - msgget(), msgctl(), msgsnd() (ipc/msg.c)
- Fichiers
  - Moyen de communication (pipe et fifo)
  - Moyen de synchro (fcntl)
- Signaux
  - Communication basique: déclenchement d'une routine dans d'autres processus
- Mémoire partagée

Périphériques

- Pilote de périphérique dialogue avec la matériel
  - ports d'entrées-sorties, in[b|w|l] / out[b|w|l]
  - registres mappés en mémoire
  - IRQ
  - DMA, Direct Memory Access
- Exemple: port série 1
  - 8 registres I/O, 0x3f8 -> 0x3f8 + 8
  - IRQ 4

Périphériques caractères

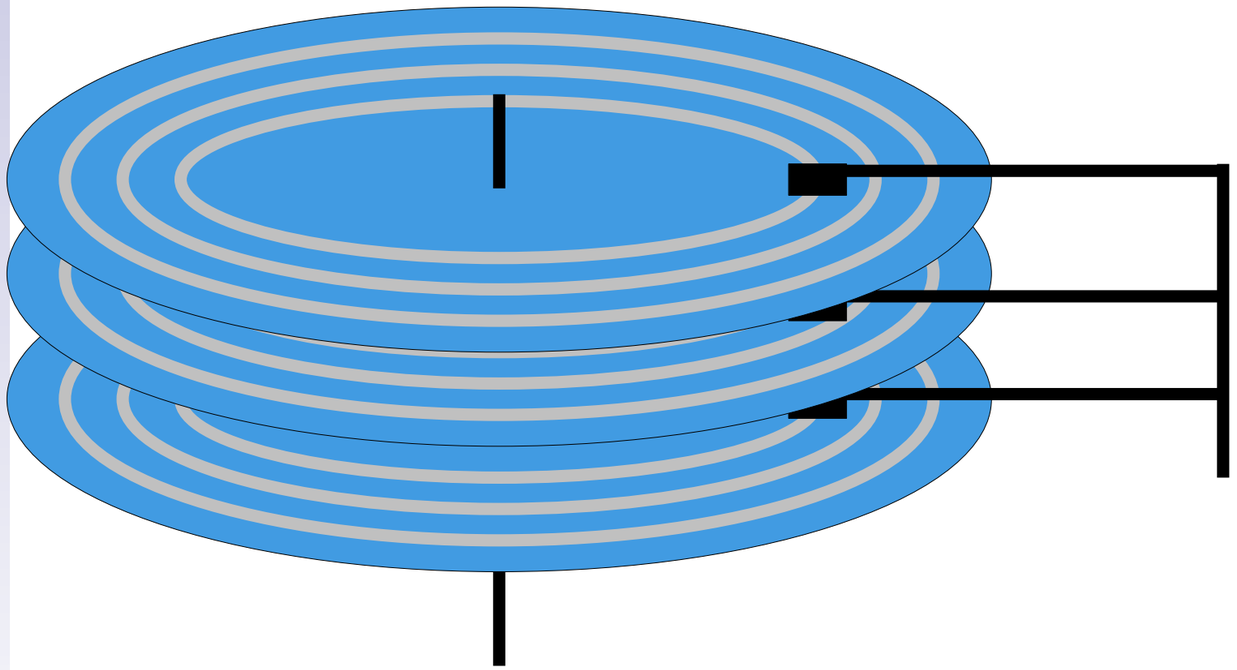
- Flux continus d'octets, pas de déplacement
- Clavier, console, ligne série, port parallèle...
- Le pilote implémente l'interface classique
  - read()
  - write()
  - ioctl()
- On peut y accéder comme un fichier (/dev)
- Pas réellement de partage des périphériques caractères

Du bloc au fichier

- Les applications n'accèdent pas à des blocs
- Elles utilisent une abstraction de plus haut niveau: le fichier
- Sous-système assez complexe
  - VFS, Virtual File System
  - pilote de système de fichiers
  - page cache
  - ordonnanceur d'I/O
  - pilote de périphérique bloc (disque, mémoire Flash...)

Un File System : pourquoi faire ???

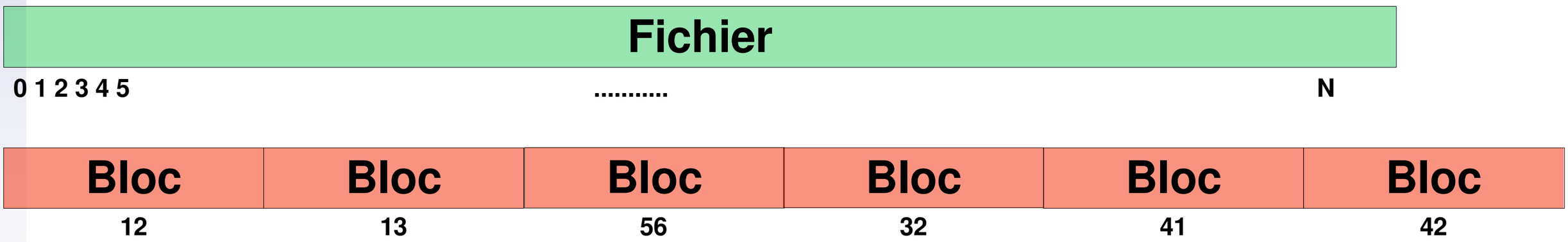
- Premier support de stockage civilisé : le disque dur !
  - Un empilement de disques (plateaux)
  - Sur chaque plateau : pistes (regroupées en cylindres)
  - Sur chaque piste : des secteurs
  - Un moteur, un bras de lecture



- 1 bloc de donnés identifié par
  - Un numéro de cylindre
  - Un numéro de tête
  - Un numéro de secteur
- Adressage simplifié et uniforme
  - 1 bloc est identifié par
    - Un numéro de bloc !!!
    - Translation physique faite par le disque

Un File System : pourquoi faire ??? (2)

- Ce que veut stocker l'utilisateur
  - Un ensemble d'octets de taille quelconque
- Ce que propose les disques
  - Un ensemble de petits blocs de taille fixe
- Abstraction de stockage proposée : le fichier
  - Une séquence d'octets de taille quelconque
  - Identifié par un nom
- Problème : faire correspondre l'abstraction à la réalité matérielle



Un File System : pourquoi faire ??? (3)

- Rôle du système de fichiers
  - Offrir l'abstraction simple de « fichier »
  - Masquer les supports de stockage
    - Complexité d'organisation et d'accès au disque
    - Diversité des tailles, organisation physique, etc...
  - Gérer au mieux l'utilisation du support de stockage
    - Éviter le gaspillage d'espace disque
    - Optimiser les accès au disque (limiter les déplacements du bras de lecture)
- Offrir une interface d'accès et de manipulation simple
  - Ouvrir / fermer
  - Lire / écrire
  - Se déplacer
  - ...

Partie 1

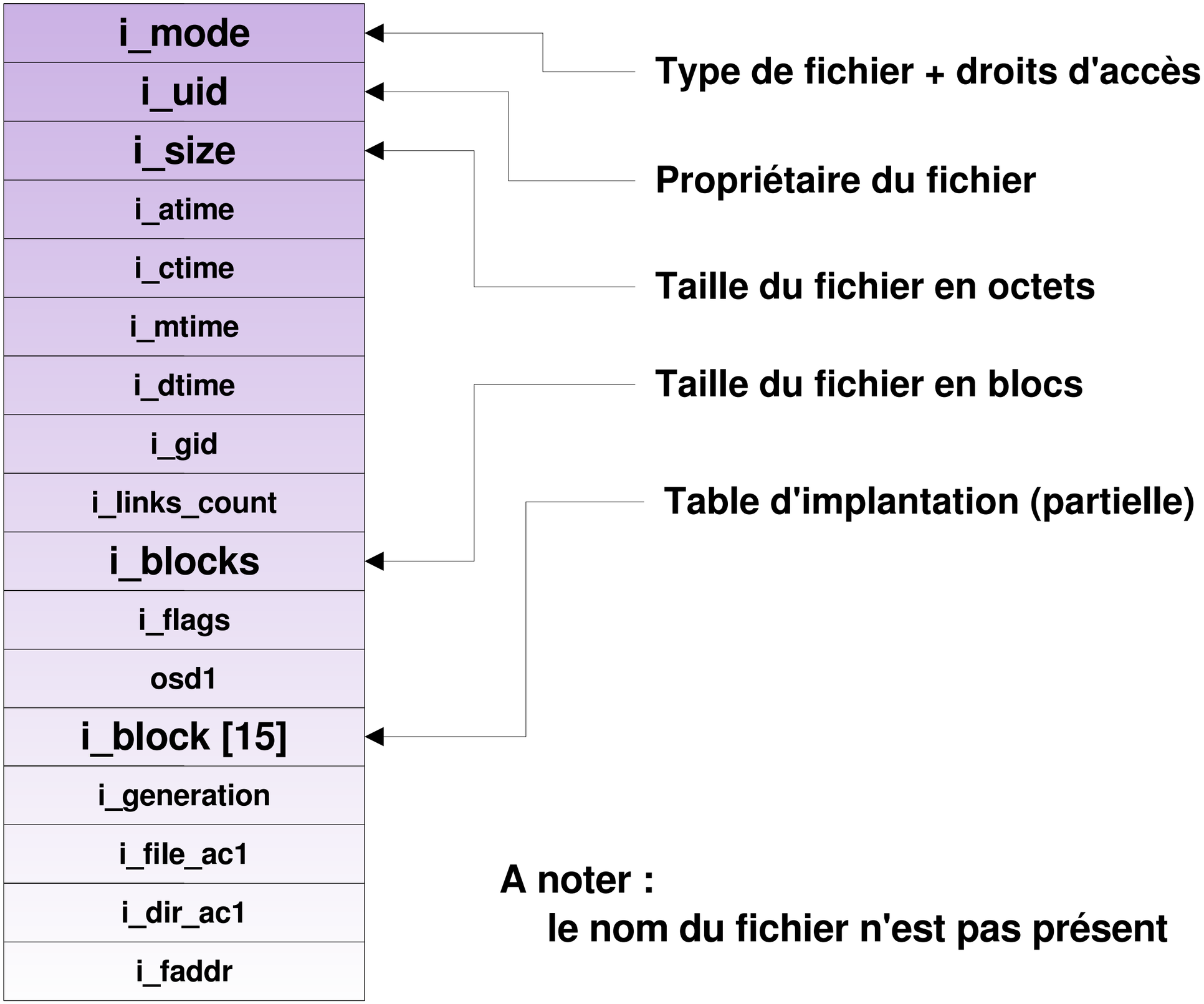
Représentation d'un fichier sur disque

Représentation d'un fichier sur disque

- Un fichier contient
  - Des données
  - Des méta-données (taille, propriétaire, date de création, ...)
- Sur disque
  - Des blocs de données
  - Des blocs de méta-données
    - Information sur le fichier (taille, propriétaire, ...) : les **i-noeuds**
    - Localisation des blocs de données sur disque : **table d'implantation**

Les i-noeuds : l'exemple d'Ext2

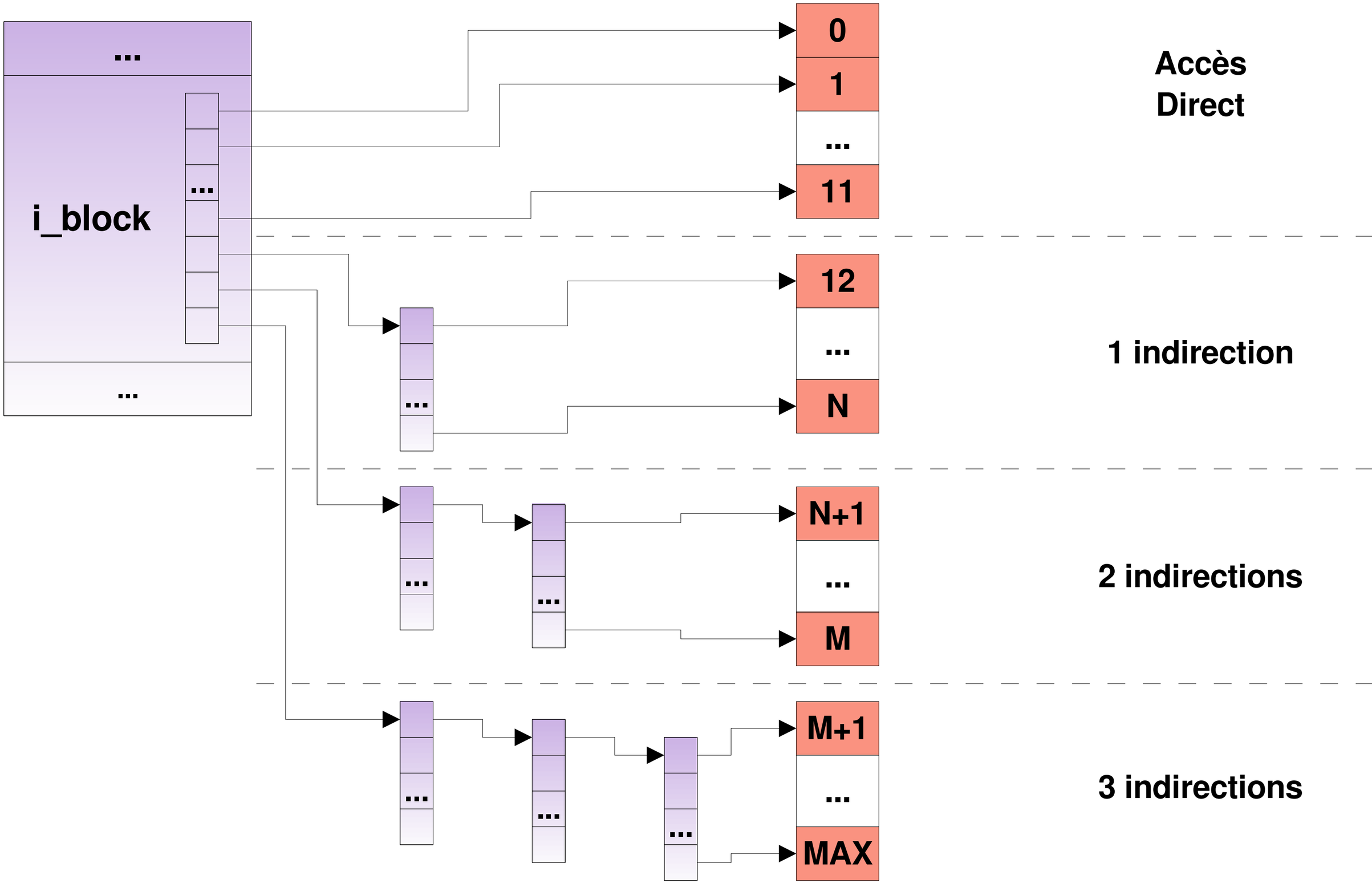
128 octets



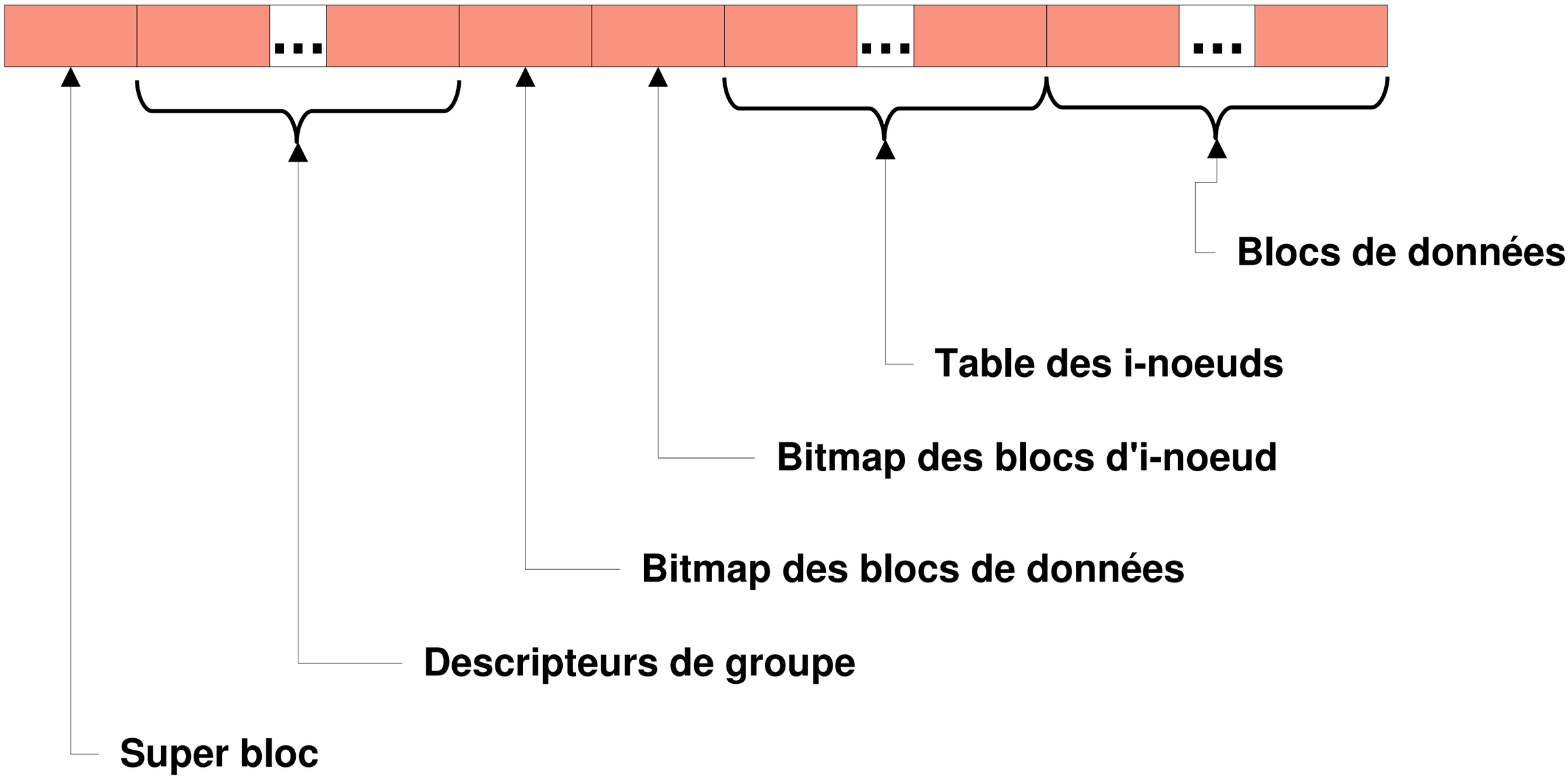
A noter :

le nom du fichier n'est pas présent

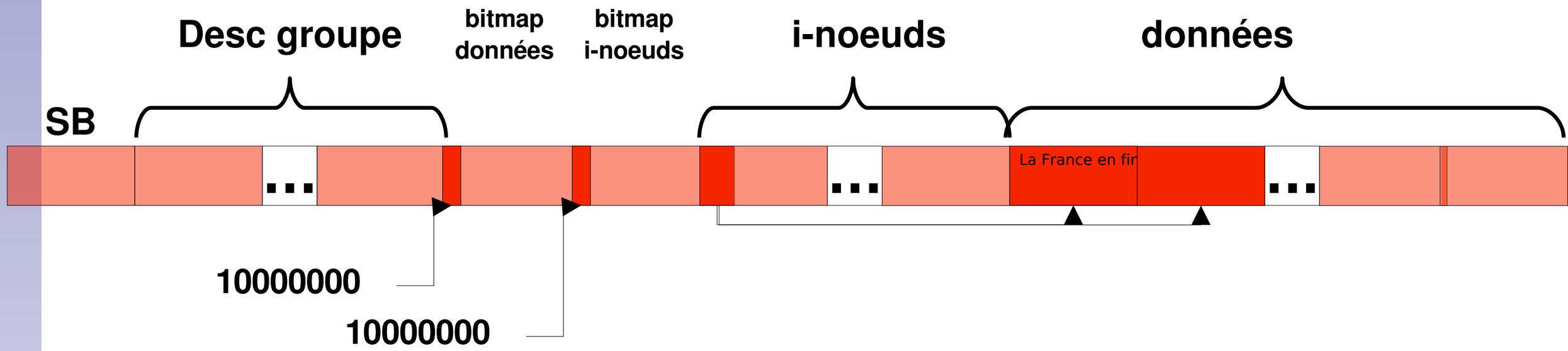
Table d'implantation : l'exemple d'Ext2



Exemple de stockage sur disque : Ext2

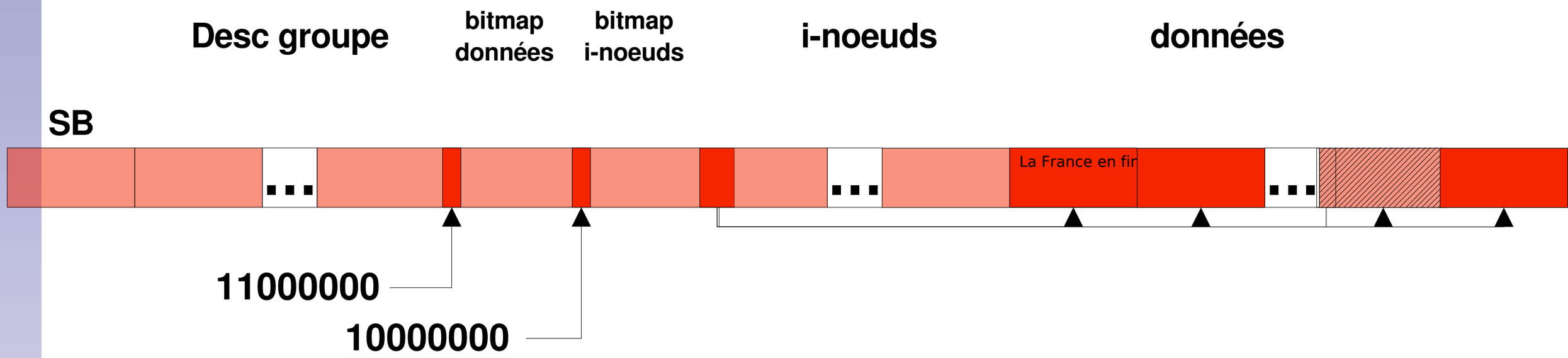


Exemple de stockage sur disque : Ext2 (cont'd)




```
dagon$ touch toto
dagon$ echo "La France en finale" >> toto
dagon$ cat /dev/random >> toto
```

Exemple de stockage sur disque : Ext2 (cont'd)

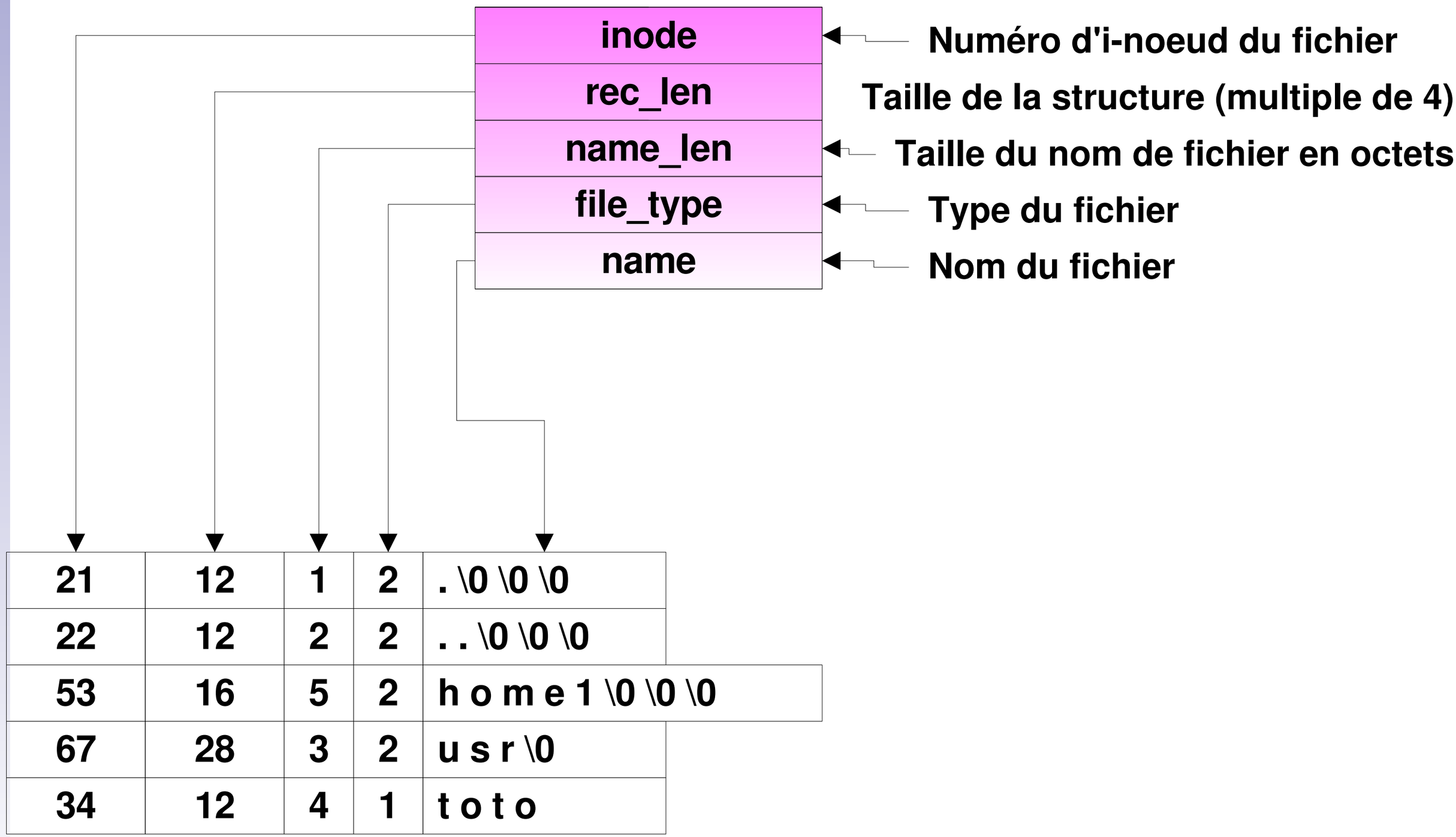



```
dagon$ touch toto
dagon$ echo "La France en finale" >> toto
dagon$ cat /dev/random >> toto
```

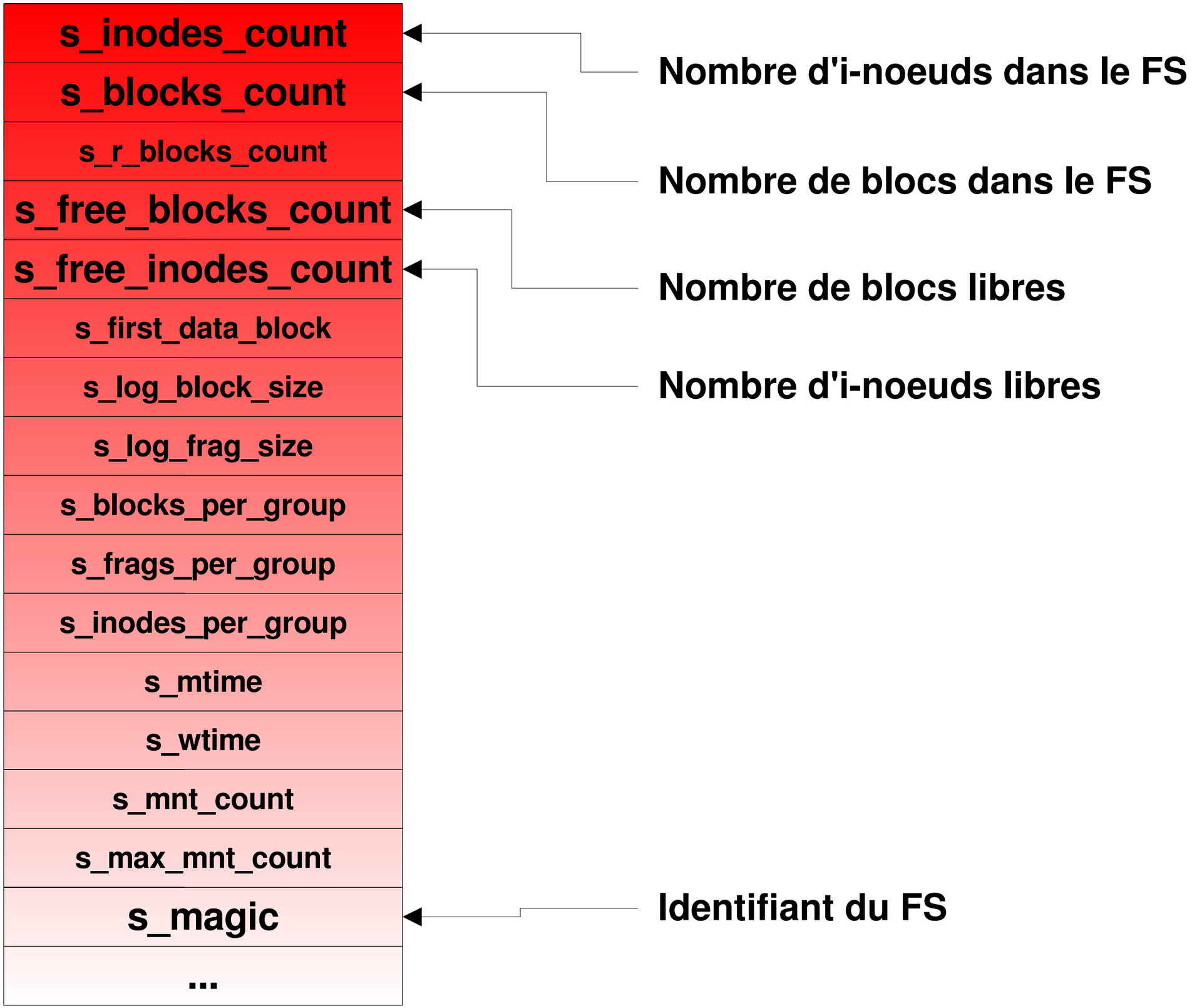
Dit, c'est quoi ton nom ?

- Un fichier a un nom...
  - toto
- ... et un chemin d'accès
  - /tmp/toto
- Le nom est stocké dans un répertoire
  - Liaison entre « nom de fichier » et « numéro d'i-noeud »
  - Lien hard : plusieurs noms sur le même i-noeud
- Un répertoire est généralement stocké dans un fichier
  - Stockage et gestion identique aux fichiers réguliers
  - Fichier « marqué » comme *répertoire*
- Le chemin d'accès n'est stocké nul part
  - Implicitement stocké dans la succession de répertoires à parcourir pour atteindre le fichier
  - Le chemin n'est pas unique

Répertoires : l'exemple d'Ext2



Super Bloc : l'exemple d'Ext2



Un VFS

Pour quoi faire ?

Systèmes de fichiers : l'embarras du choix

- Systèmes de fichiers « classiques »
  - FAT (16/32/ VFAT)
  - NTFS
  - Unix FS
  - Minix FS
  - Ext2 / 3
  - Reiser FS
  - ...

Systèmes de fichiers : l'embarras du choix (2)

- Systèmes de fichiers distribué
  - NFS
  - AFS
  - Samba
  - Coda
  - GFS (?)
  - ...
- Systèmes de fichiers « virtuels »
  - /proc
  - sysfs
  - ...

Unix : le tout fichier !

- Unix dit : tout est fichier !
  - Supports de stockage
  - Communications (pipes, sockets)
  - Périphériques (clavier, imprimante, ...)
  - etc...

Systèmes de fichiers : l'embarras du choix (3)

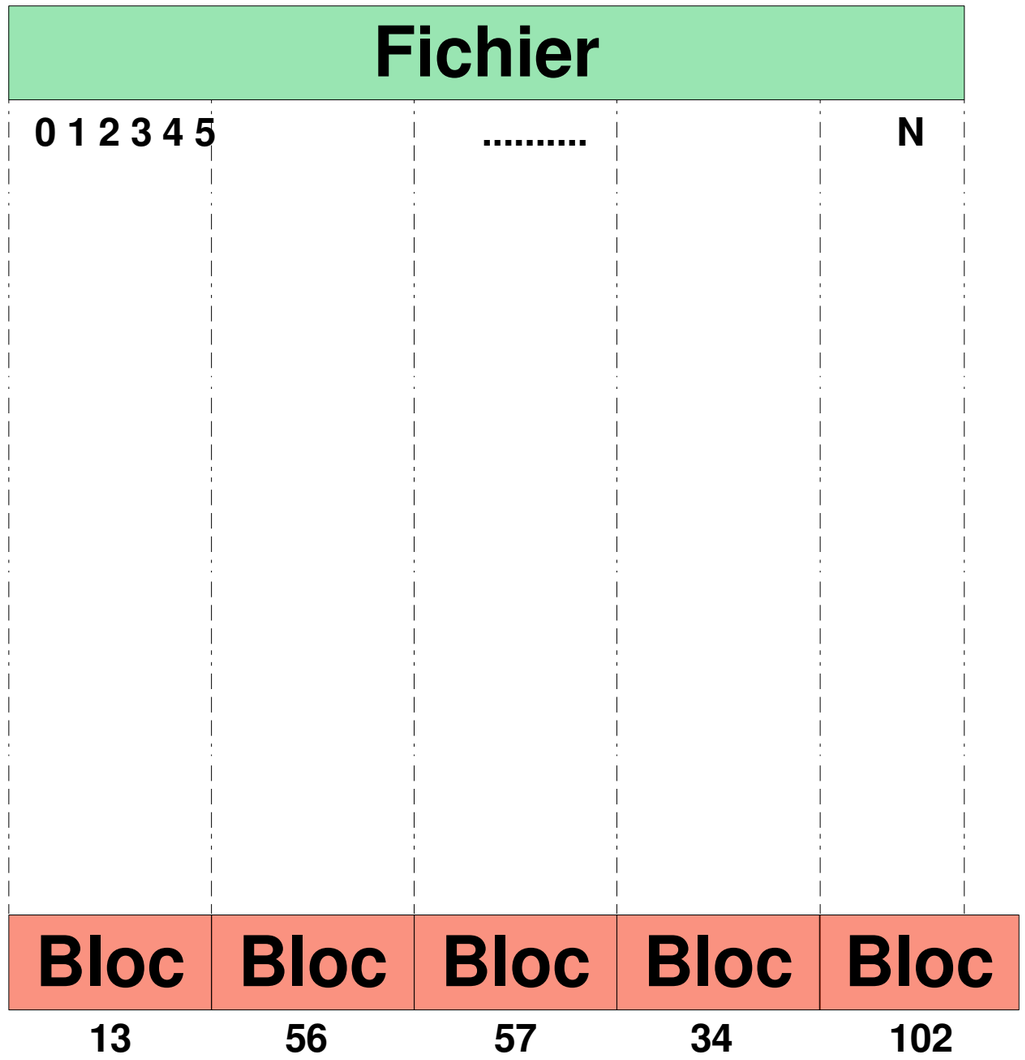
- Systèmes de fichiers « spéciaux »
  - sock_fs
  - pipe_fs
  - mqueue_fs
  - ...
- Mais... en prenant un peu de recul
  - Tous ces systèmes ont un air de ressemblance !

Architecture d'un système de fichiers « classique »

Interface

(open, read, write, close, seek, ...)

Gestionnaire de blocs



Les caches

- Un accès au disque est très coûteux
  - Latence : ~ 5 ms
  - Débit : ~ 20 - 80 Mo / s
- Accès à la mémoire : très rapide
  - Latence : 1000 fois inférieur
  - Débit : 1000 fois supérieur
- De nombreux caches dans la plupart des FS
  - Cache de données
  - Cache d'i-noeuds
  - Cache de répertoires
  - Cache de blocs
  - ...

Architecture d'un système de fichiers « classique »

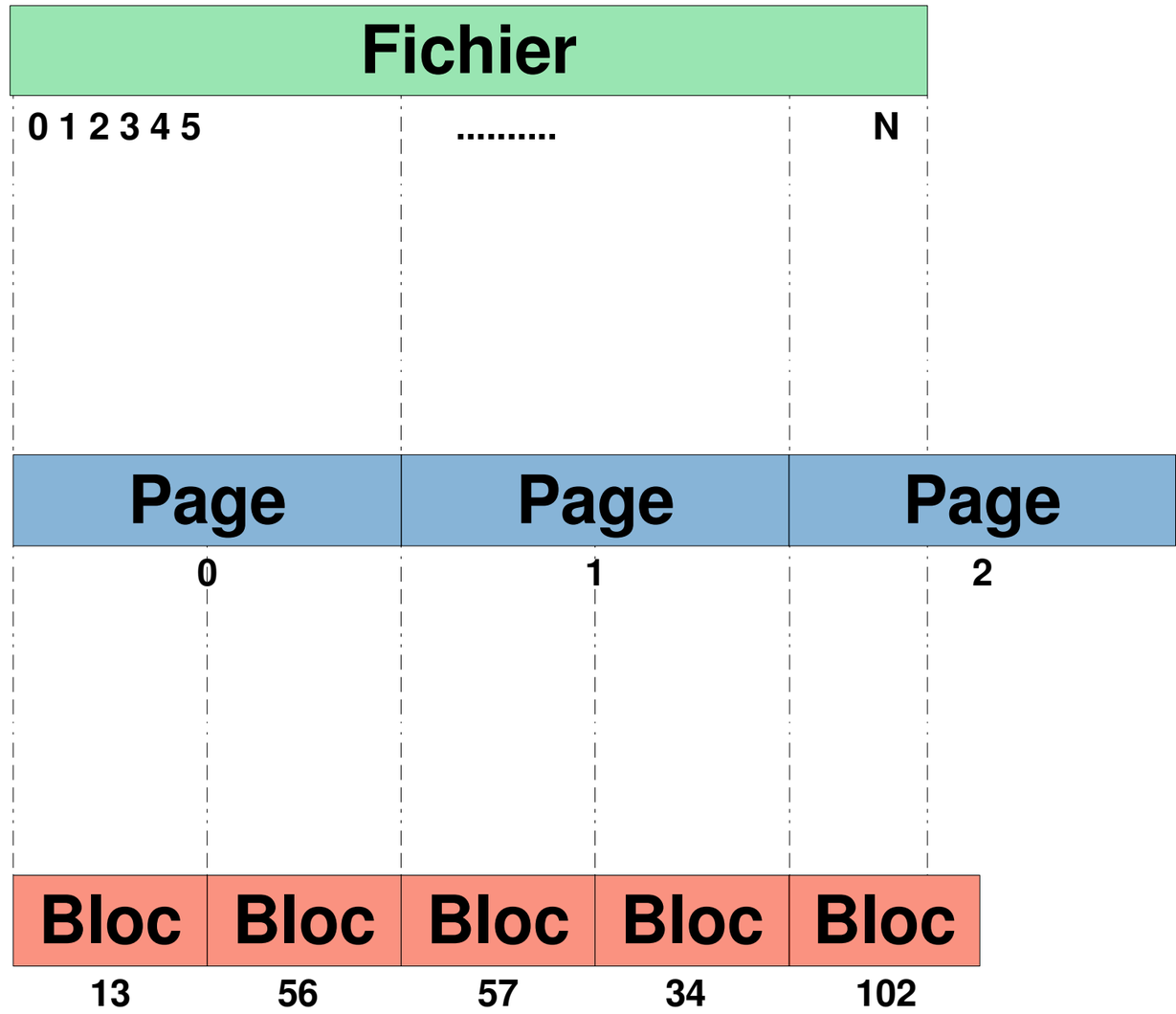
Interface

(open, read, write, close, seek, ...)

Caches

Données, i-noeuds, répertoires, ...

Gestionnaire de blocs



Architecture d'un système de fichiers distribué

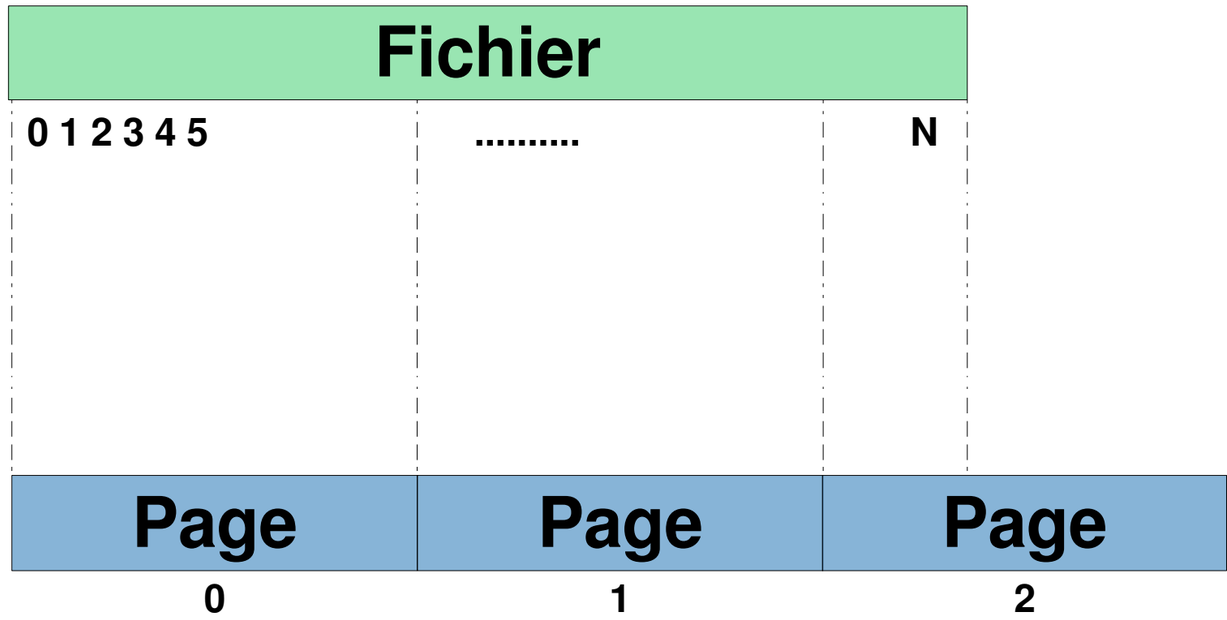
Interface

(open, read, write, close, seek, ...)

Caches

Données, i-noeuds, répertoires, ...

Client du FS



Architecture d'un système de fichiers « spécial »

Interface

(open, read, write, close, seek, ...)

Fichier

0 1 2 3 4 5 ..... N

Caches

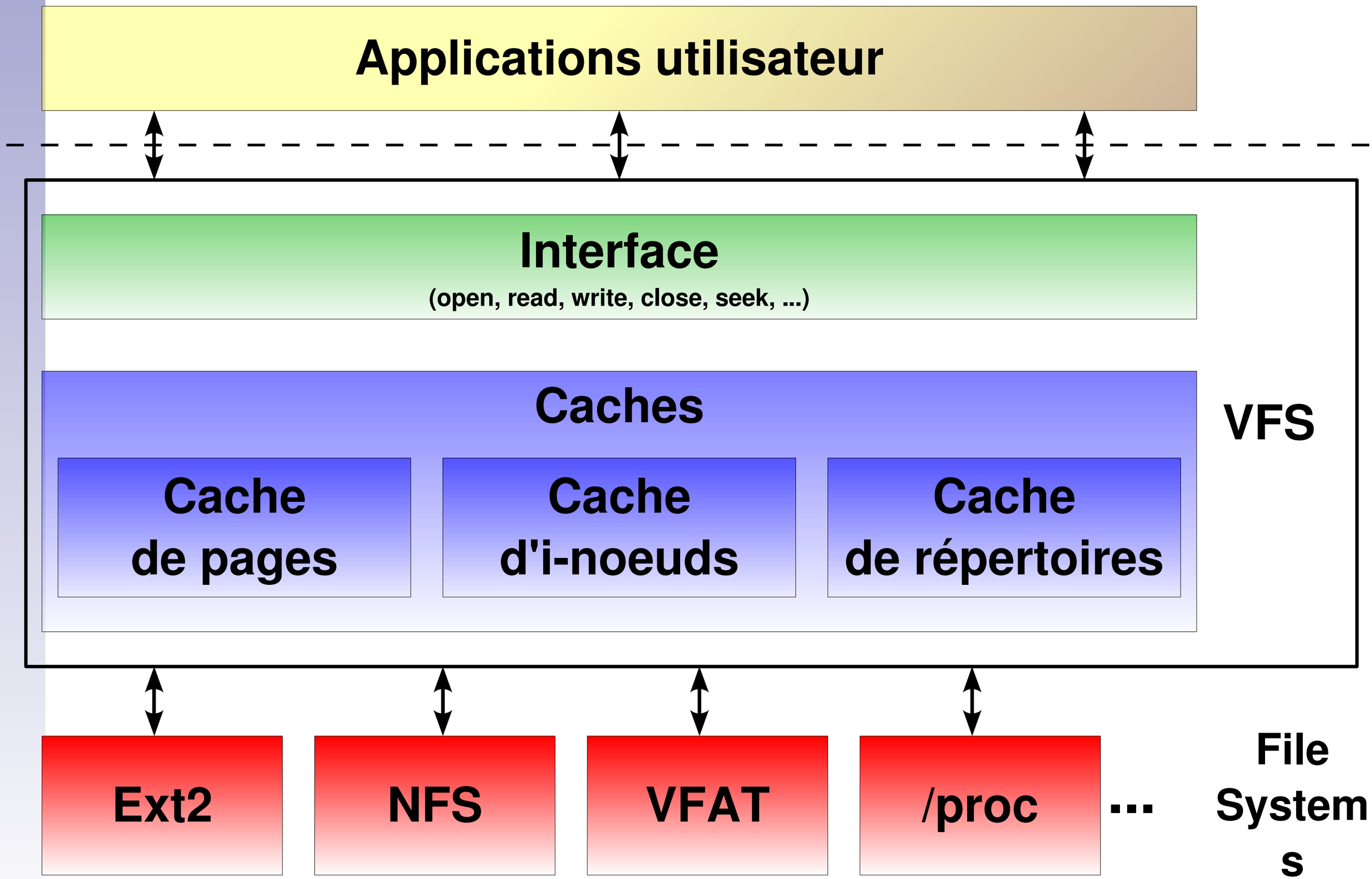
i-noeuds, répertoires, ...

Gestionnaire de FS « spécial »

Le VFS

- Objectif
  - Offrir une interface unique aux utilisateurs
  - Factoriser l'architecture et le code des différents FS
    - Une architecture générique sur laquelle sont construit les «vrais» FS
  - Offrir une arborescence unique pour
- Une mise en oeuvre « orientée objet »
  - Un ensemble d'objets (fichiers, i-noeuds, ...)
  - Un ensemble de méthodes pour manipuler chaque type d'objet
  - Le noyau est écrit en C : émulation du comportement objet
    - Association objet
    - Instantiation
    - Héritage
    - ...

Architecture du VFS

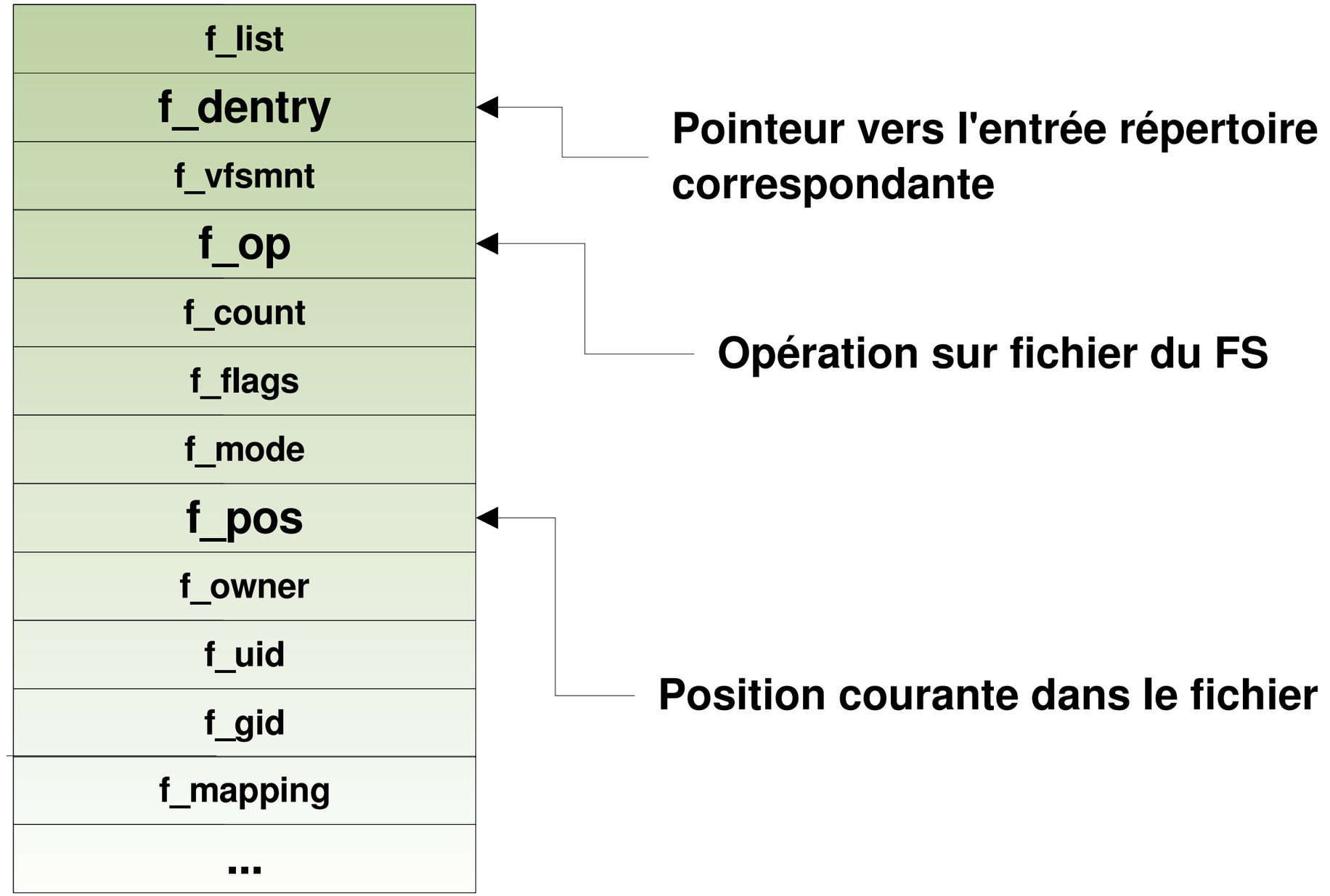


Structures de données du VFS

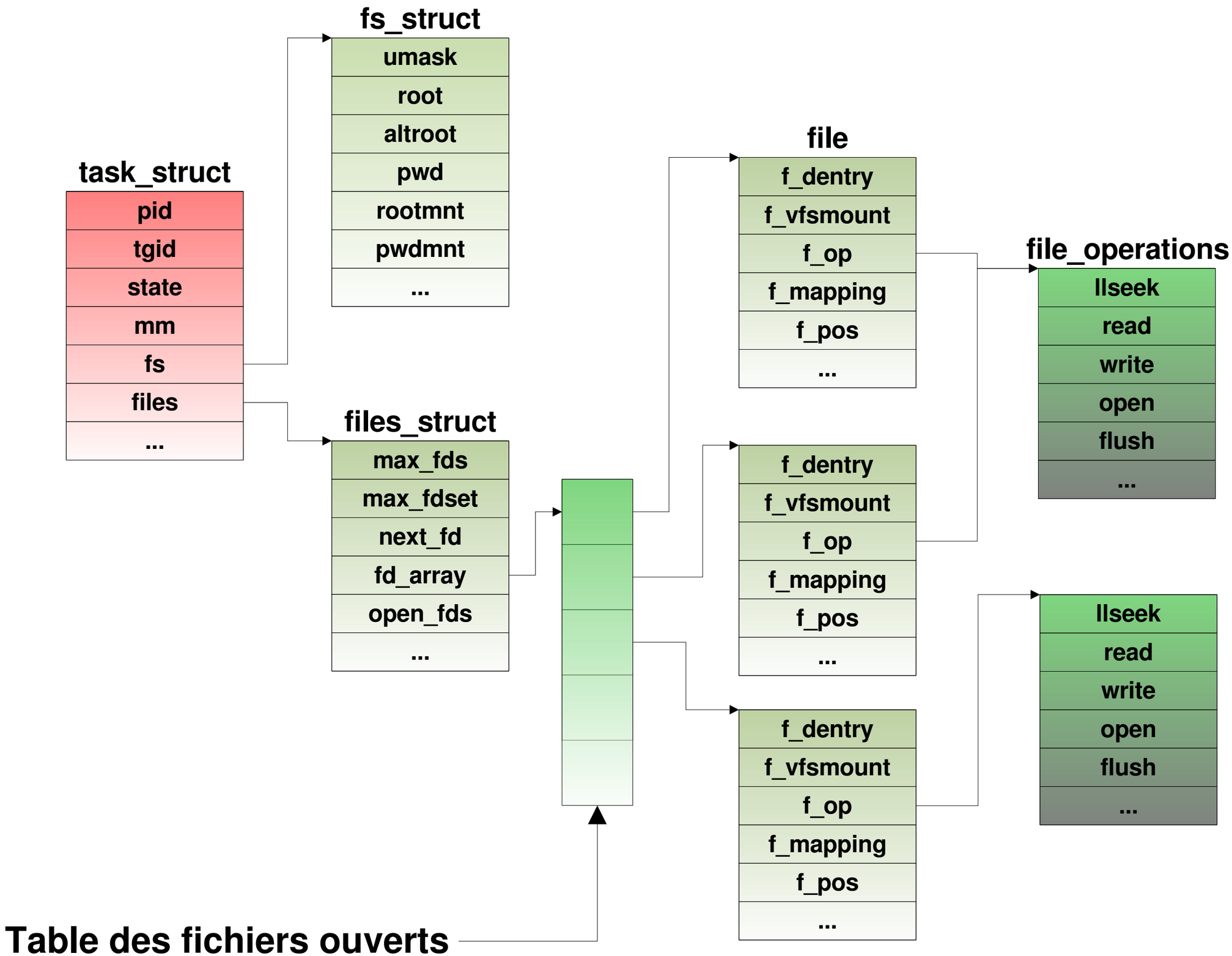
- 2 grands types de structures
  - Données
    - Générique
    - Commun à tous les FS
    - Exemple
      - File
      - inode
      - dentry
      - ...
  - Opérations sur les données
    - Spécialisation pour chaque FS
      - file_operations
      - inode_operations
      - dentry_operations
      - ...

File struct

- Représente un fichier ouvert
- Une structure par fichier ouvert

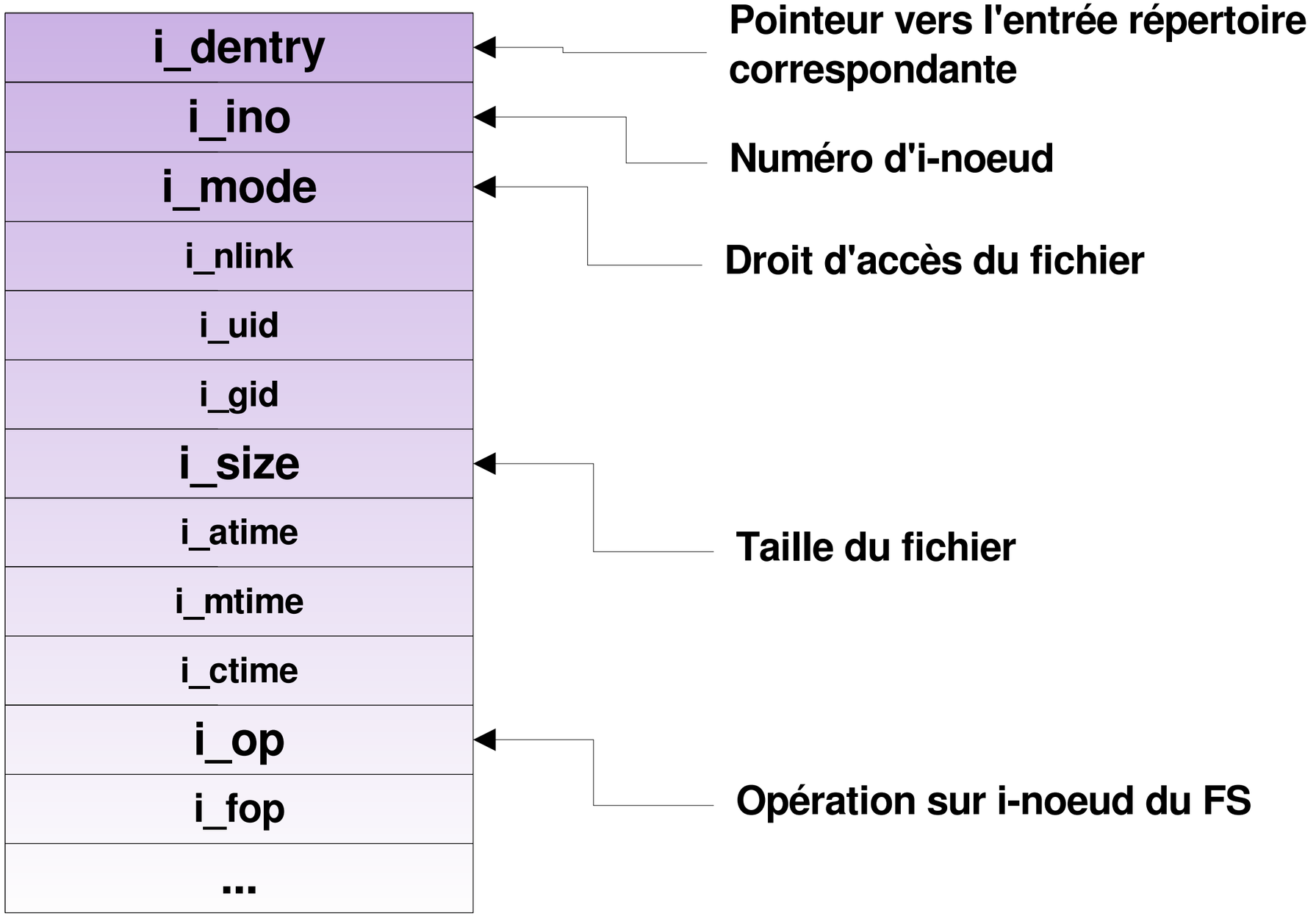


File struct


fs_struct
umask
root
altroot
pwd
rootmnt
pwdmnt
...
task_struct
pid
tgid
state
mm
fs
files
...
files_struct
max_fds
max_fdset
next_fd
fd_array
open_fds
...
file
f_dentry
f_vfsmount
f_op
f_mapping
f_pos
...
file_operations
llseek
read
write
open
flush
...
f_dentry
f_vfsmount
f_op
f_mapping
f_pos
...
llseek
read
write
open
flush
...
f_dentry
f_vfsmount
f_op
f_mapping
f_pos
...
Table des fichiers ouverts

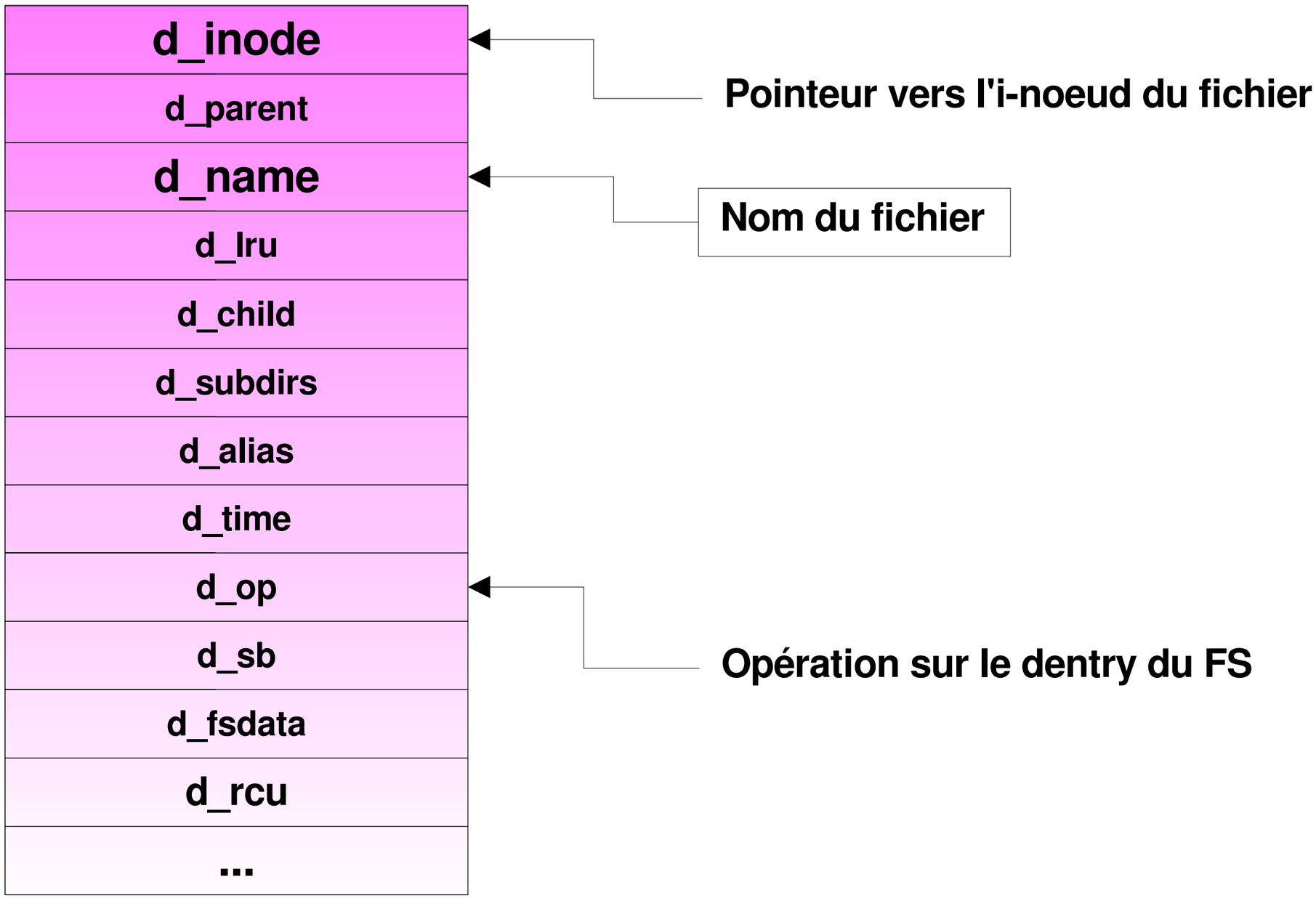
Inode struct

- Stocke en mémoire un i-noeud physique
- Une structure par fichier physique



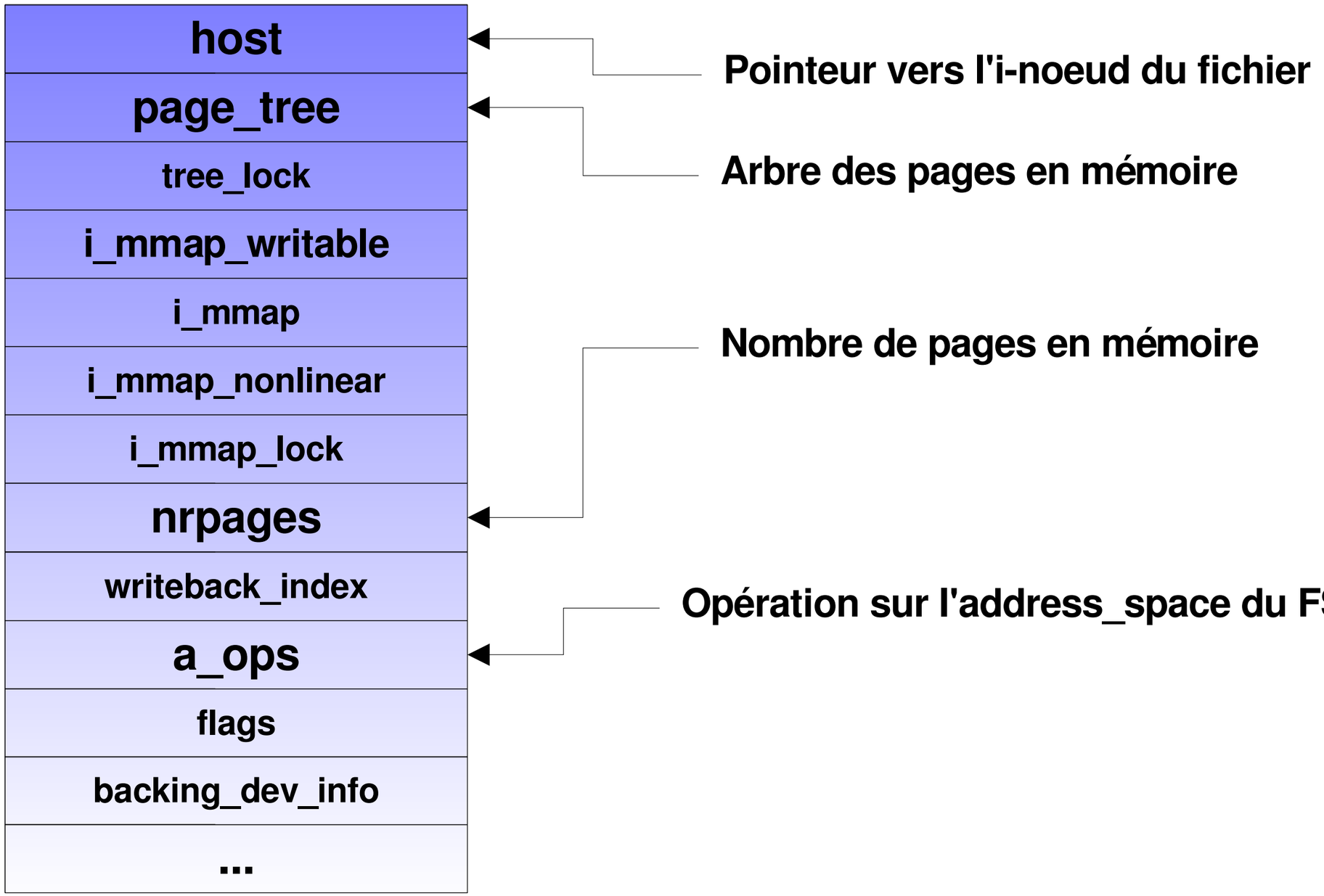
Dentry struct

- Stocke en mémoire une entrée d'un répertoire
- Une structure par entrée de répertoire

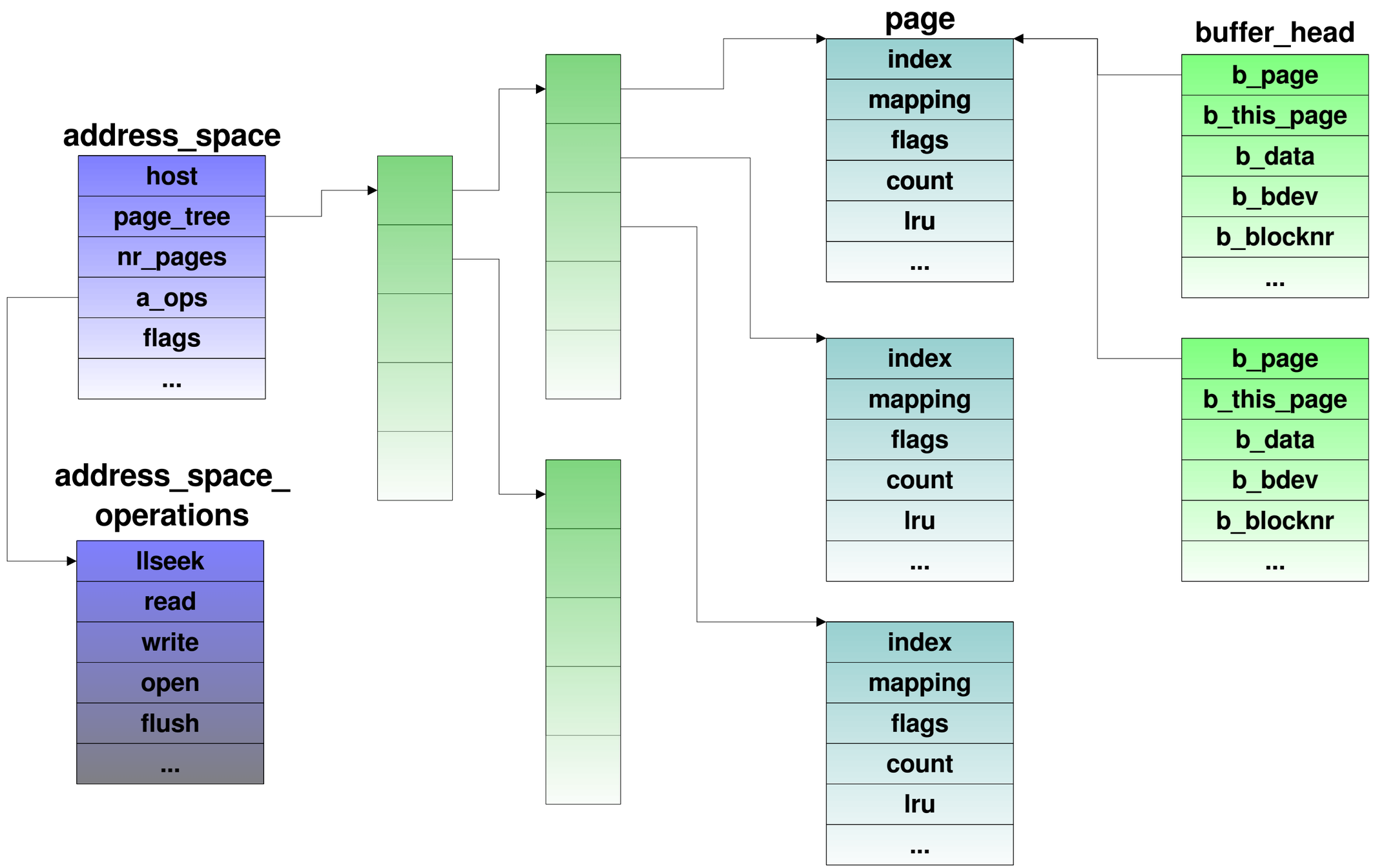


Address_space struct

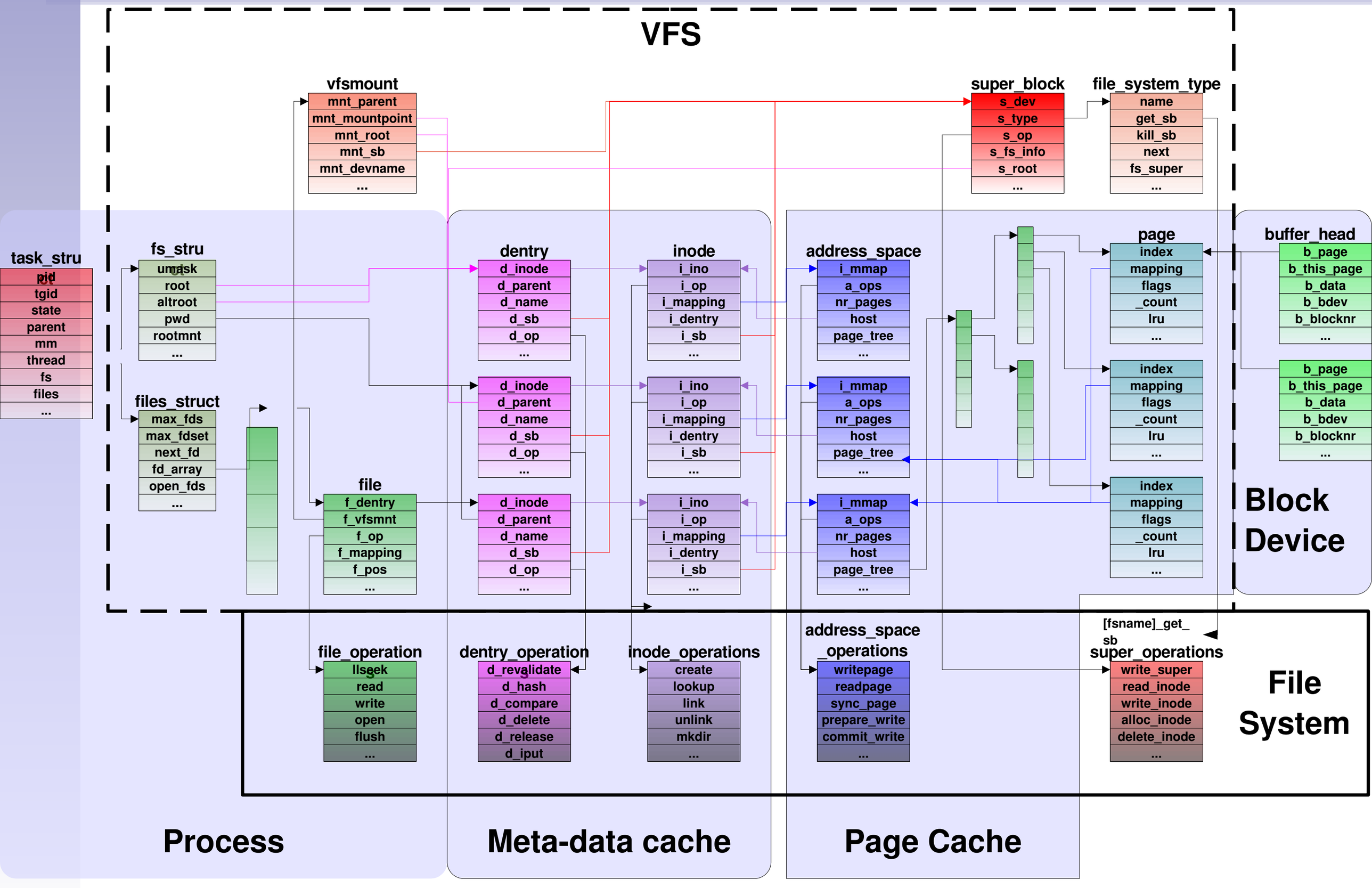
- Représente le cache de données d'un fichier
- Une structure par fichier physique



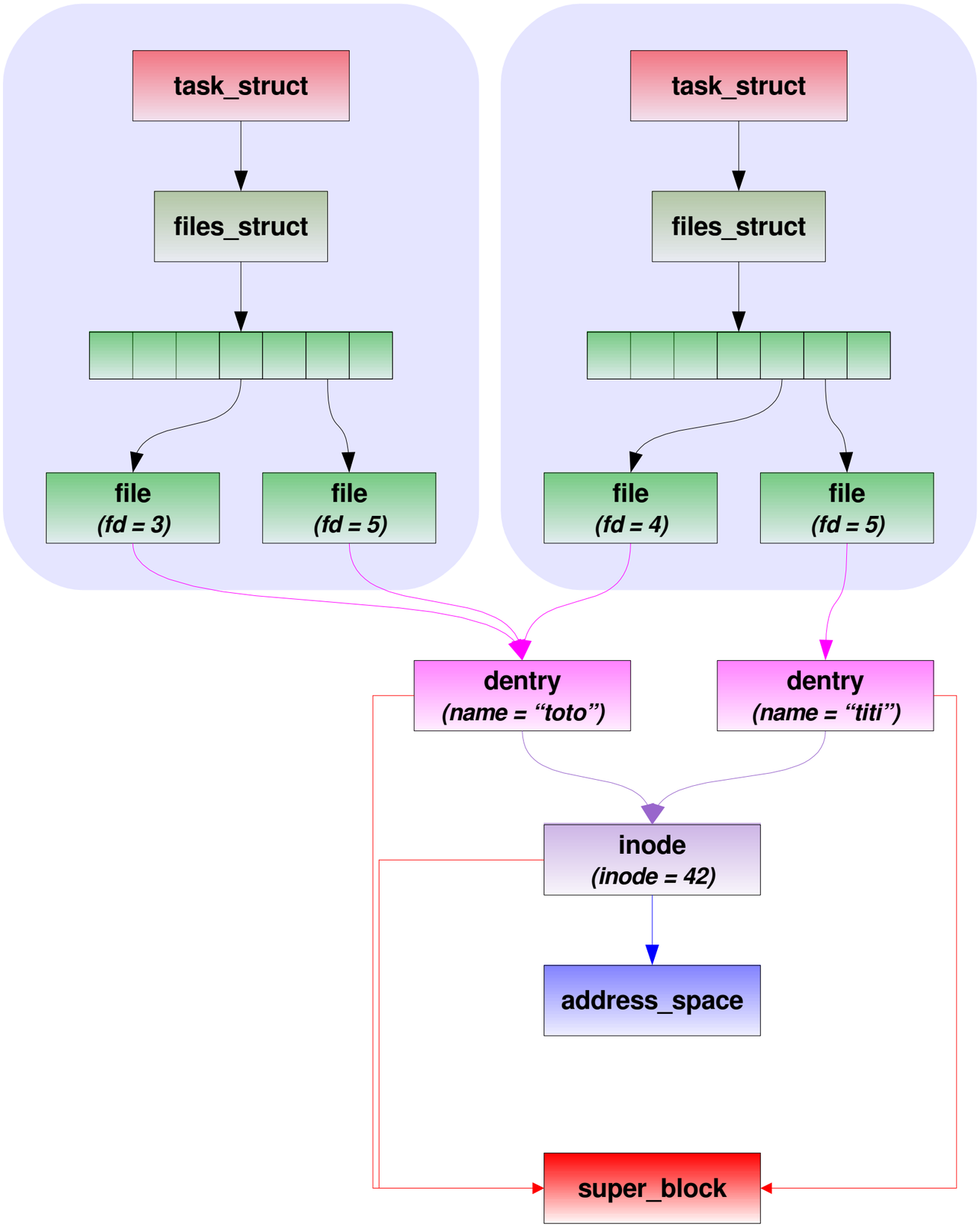
Address_space struct



VFS Big Picture



Un petit exemple



Partie 3

Le VFS par l'exemple

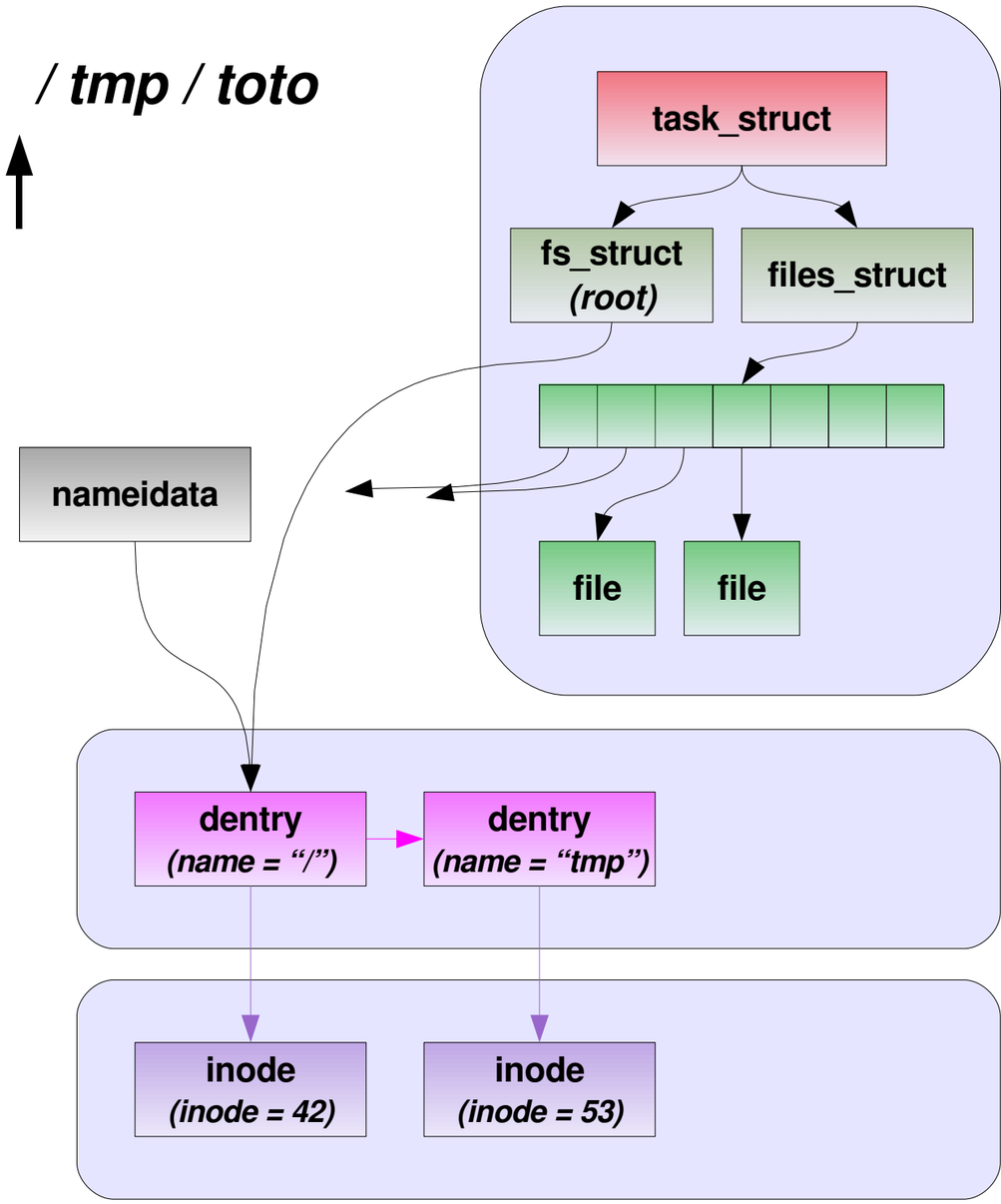
La fonction « *open* »

fd = open("/tmp/toto")

sys_open

- **get_unused_fd**
- **filp_open**
  - **open_namei**
    - **path_lookup**

VFS | **Ext2**

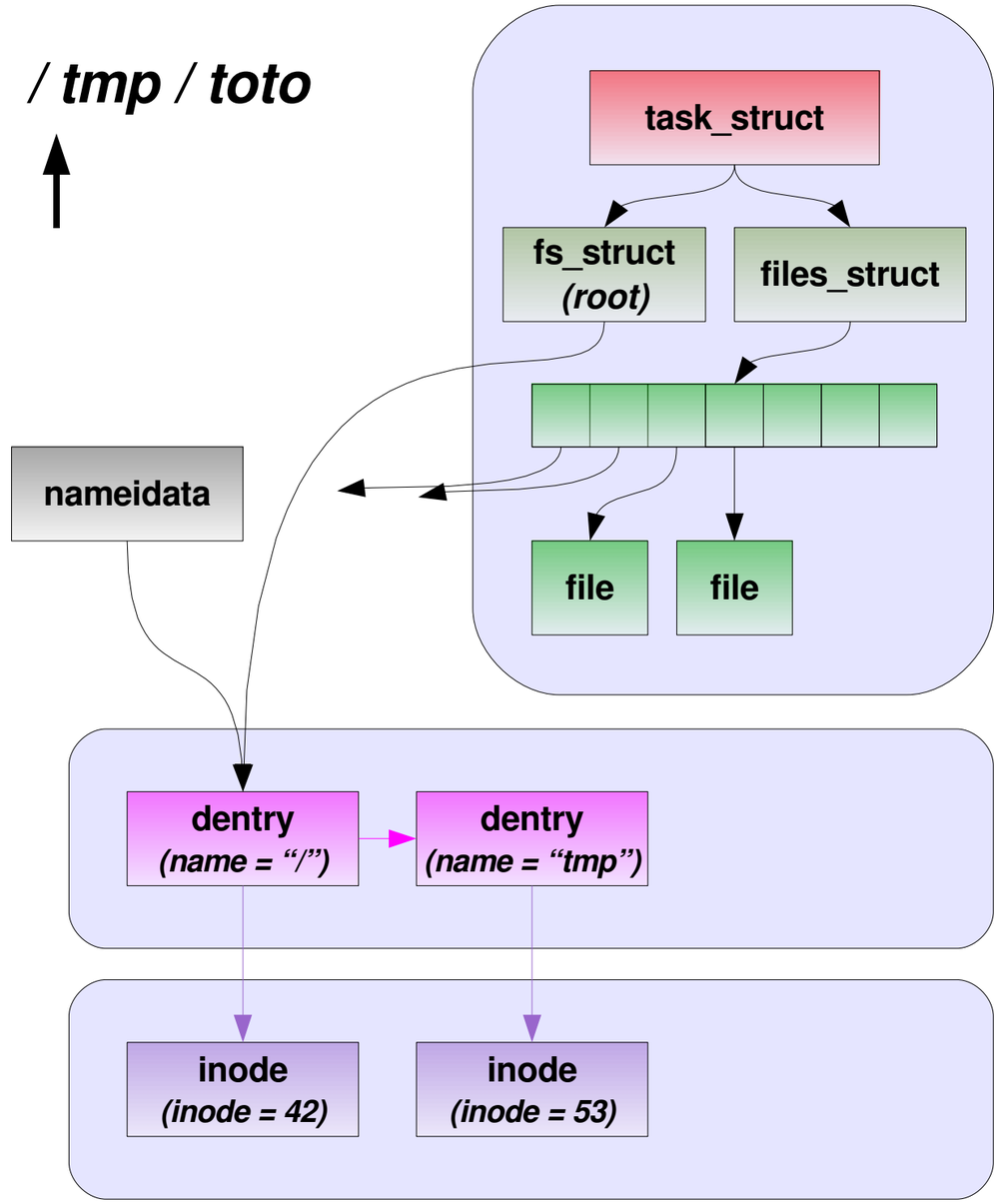


La fonction « *open* »

fd = open("/tmp/toto")

VFS | **Ext2**

- **sys_open**
  - **get_unused_fd**
  - **filp_open**
    - **open_namei**
      - **path_lookup**
        - **link_path_walk**
          - **do_lookup**
            - **__d_lookup**

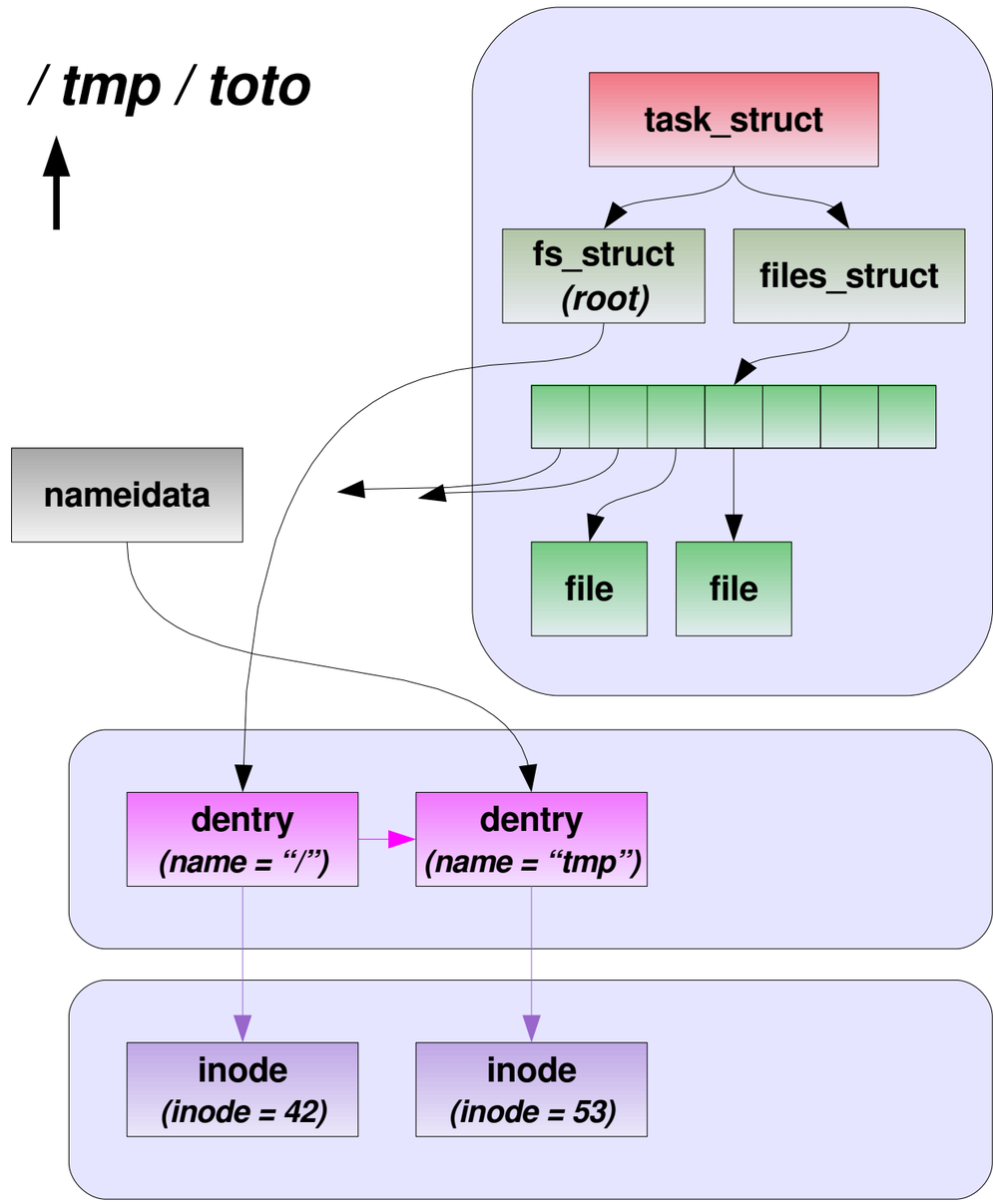


La fonction « *open* »

fd = open("/tmp/toto")

VFS | **Ext2**

- **sys_open**
  - **get_unused_fd**
  - **filp_open**
    - **open_namei**
      - **path_lookup**
        - **link_path_walk**
          - **do_lookup**
            - **__d_lookup**

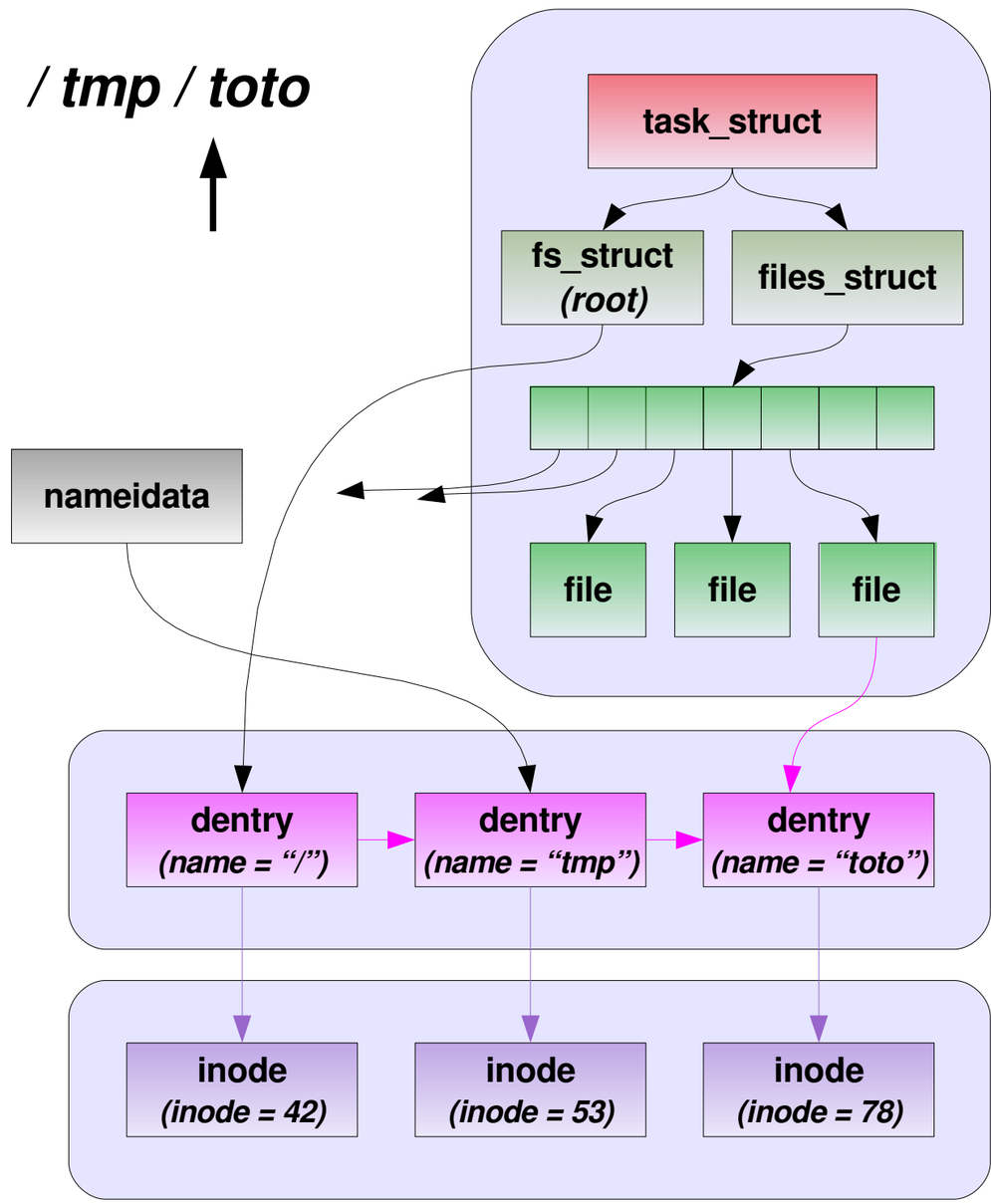


La fonction « *open* »

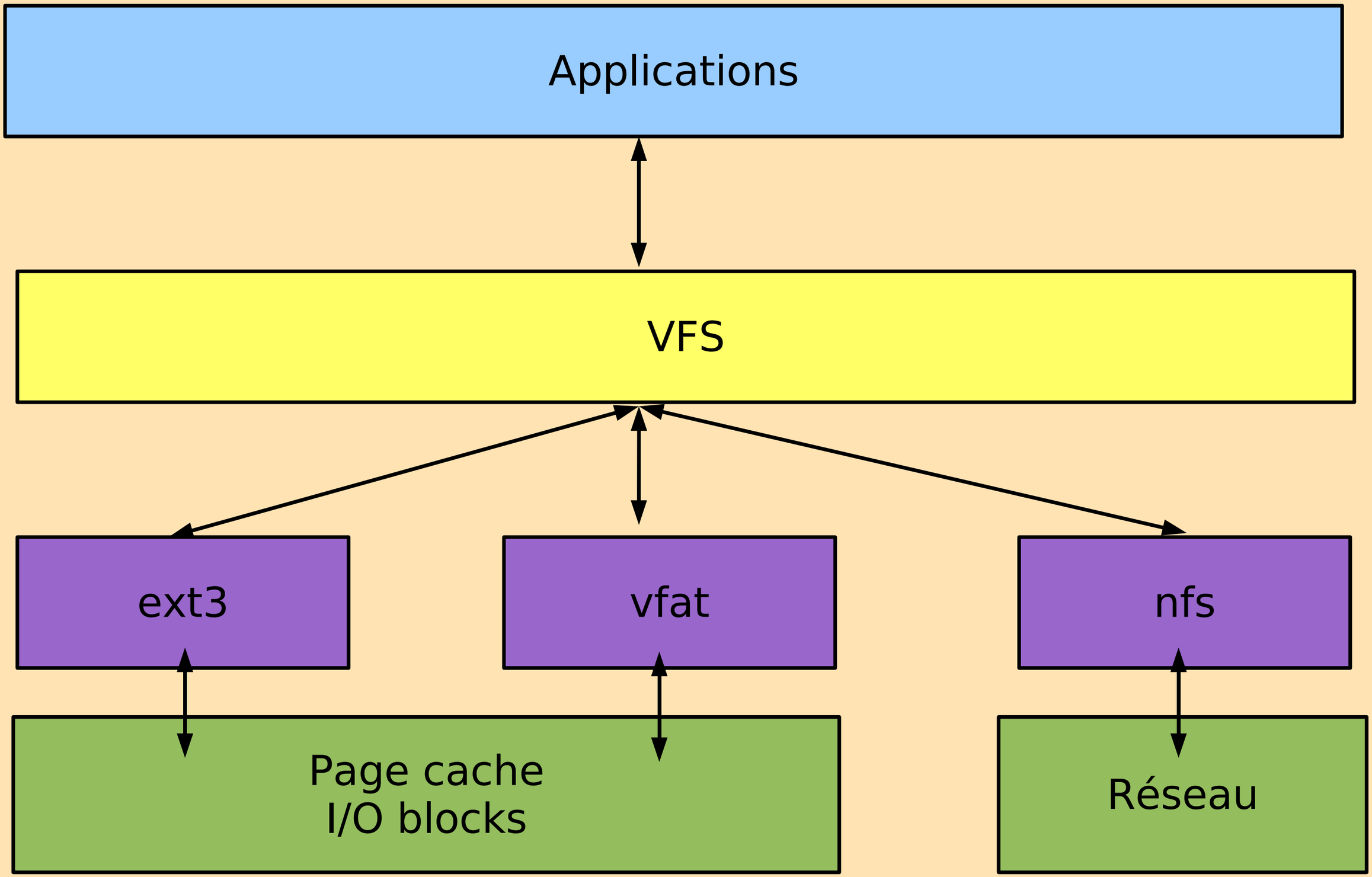
fd = open("/tmp/toto")

VFS | **Ext2**

- **sys_open**
  - **get_unused_fd**
  - **filp_open**
    - **open_namei**
      - **path_lookup**
        - **link_path_walk**
          - **do_lookup**
            - **__d_lookup**
            - **real_lookup**
              - **d_alloc**
              - **dir->i_op->lookup** — **ext2_lookup**
    - **may_open**
  - **dentry_open**
    - **get_empty_filp**
    - **f->f_op->open**
      - **generic_file_open**
  - **fd_install**



Accès aux fichiers



VFS

- VFS, Virtual Filesystem
- Implémente les appels systèmes read(), write(), seek(), readdir()
  - interface générique d'accès aux fichiers
- Expose une arborescence unique de tous les fichiers
- Repose sur l'implémentation de différents *callbacks* dans chaque pilote de système de fichiers

I/O

- Page cache émet des *struct bio*
  - *(device, secteur, pages, taille, type)*
- Un disque dur est composé de mécanique, il faut ordonnancer les entrées-sorties, et les agréger pour maximiser les performances
- Les *bio* passent et attendent dans un ordonnanceur d'I/O : anticipatory, CFQ, deadline, noop
- Ils ressortent combinés sous la forme d'une *struct request*

Pilote bloc

- Le pilote de disque reçoit des *struct request*
- Les traitent de manière spécifique au matériel (commandes ATA, SCSI, ATAPI...)
- Terminaison signalée de manière asynchrone par IRQ, puis remontée

Séquencement