Opérations sur les fichiers

- Ouvertures de fichiers
- Descripteurs de fichiers
- Entrée-sortie temporisées
- Positionnement dans un fichier
- Fichiers spéciaux

Représentation des fichiers

- Tar archive
- Fichier sur le disque

Exercices

Pour vous réveiller...

1. Peut-on deviner le résultat de `access` f `W_OK` à partir des informations retournées par `lstat` f ? par `stat` f ?
2. Une hiérarchie de fichiers est-elle un arbre ? un graphe ? un graphe acyclique ? **Quelles précautions faut-il prendre lorsqu'on la parcourt ?**
3. Comment identifier le nœud racine d'une hiérarchie de fichiers.

Pour vérifier que vous êtes bien réveillés...

4. Comment renommer un fichier régulier sans utiliser l'appel système `rename` ?
5. À quoi sert rename, alors ?
6. Pourquoi n'y a-t-il pas de liens durs entre les répertoires ?

Enchevêtré dans les droits d'accès...

1. Bod exécute la commande "`cd`". Où se retrouve-t-il ?
2. Il veut effectuer "`chmod og-x •`". Que fait cette commande ?
3. Il se trompe et exécute "`chmod -x •`". Puis il fait "`ls -l •`". Ayant compris son erreur, il fait "`chmod +x •`" mais à la vue de la réponse de la machine, il s'affole. Que s'est-il passé ?
4. Sauriez-vous l'aider à réparer son erreur ?

Opérations sur les fichiers réguliers

Fichiers réguliers

- Ils contiennent une suite arbitraire d'octets.
- Ils peuvent être de taille (presque) arbitraire.

Comme pour les répertoires on y accède selon la séquence :

1. ouvrir le fichier (lecture, écriture),
2. lire/écrire des données par blocs,
3. refermer le fichier.

Fichiers spéciaux

- Unix traite autant que possible les accès aux périphériques comme des opérations sur les fichiers réguliers.
- Les mêmes appels système sont applicables aux fichiers spéciaux, mais certains peuvent être inopérants.
- D'autres appels système leur sont spécifiques.

Ouverture d'un fichier


```
openfile: string -> open_flag list -> file_perm -> file_descr
```

où `open_flag list` décrit les modes d'ouverture

1. **doit** contenir exactement un des trois drapeaux suivants :

`O_RDONLY`	ouverture en lecture seule
`O_WRONLY`	ouverture en lecture seule
`O_RDWR`	ouverture en lecture et en écriture

2. **peut** contenir un ou plusieurs autres drapeaux parmi :

`O_APPEND`	ouverture en ajout
`O_CREAT`	créer le fichier s'il n'existe pas
`O_TRUNC`	tronquer le fichier à zéro s'il existe déjà
`O_EXCL`	échouer si le fichier existe déjà

(+ autres drapeaux — voir poly)

Exemples

- Lecture d'un fichier (qui doit exister) :

```
openfile filename [ O_RDONLY ] 0
```

- Écriture d'un fichier (avec écrasement si existant) :

```
openfile filename [ O_WRONLY; O_TRUNC; O_CREAT ] 0o666
```

- Idem pour un fichier exécutable (code compilé, shell-script) : remplacer 0o666 par 0o777.
- Idem, pour un fichier confidentiel : remplacer 0o666 par 0o600.
- Écriture à la fin d'un fichier existant :

```
openfile filename [ O_WRONLY; O_APPEND ] 0o666
```

Atomicité

Création d'un fichier

Si plusieurs processus effectuent en parallèle l'appel système :

```
openfile filename [ O_RDONLY; O_EXCL; O_CREAT ] 0o666
```

un seul au plus peut réussir.

Les fichiers peuvent être utilisés comme verrous : un processus prend le verrou avec la commande précédente, effectue ses tâches de façon exclusive, puis libère le verrou en détruisant le lien avec `unlink`.

Mode append

Si plusieurs processus ouvrent en parallèle le même fichier en mode `O_APPEND` et les écritures sont entrelacées, chaque écriture sera effectuée à la fin du fichier, donc aucune ne sera écrasée.

Descripteurs de fichiers

- Un descripteur est une structure allouée par le système qui mémorise les informations sur les droits de lecture-écriture le mode d'écriture, la position courante dans le fichier.
- L'ouverture d'un fichier retourne un descripteur de fichier.
- Les descripteurs sont un mécanisme général pour la lecture-écriture en Unix : ils peuvent être ouverts sur d'autres objets que les fichiers réguliers : fichiers spéciaux, tuyaux, prises.
- Un processus est lancé avec trois descripteurs pré-alloués `stdin`, `stdout` et `stderr` (de la librairie `Unix`) — voir cours sur les processus.

Remarque Un descripteur ouvert peut parfois être utilisé comme racourci pour désigner un fichier : `fstat`, `ftruncate`, `fchmod`, `fchown`.

Fermeture d'un descripteur

Il faut libérer les descripteurs dès qu'il ne sont plus utilisés pour éviter les fuites de mémoire (au niveau du programme).

```
close : file_descr -> unit
```

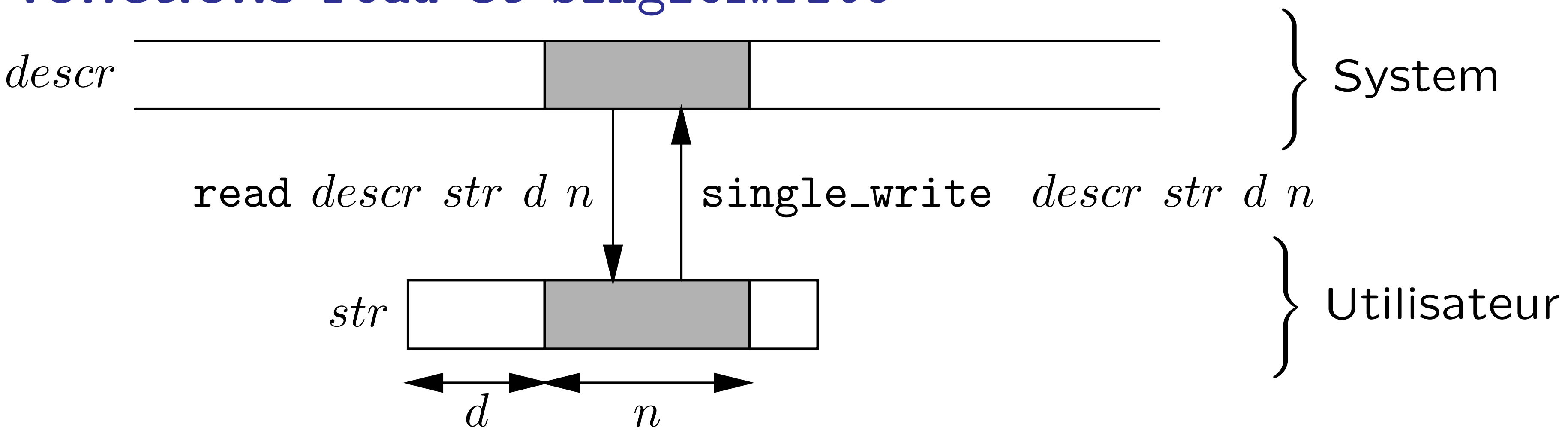
L'utilisation d'un descripteur fermé provoquera une erreur.

Bon conseils

- **Si possible** la fonction qui ouvre un descripteur le referme.
- **Attention aux exceptions !** non fatales qui doivent être attrapées pour refermer le descripteur, puis relancées
  (voir cours précédent).

  ```
  let desc = openfile ... in
  let process desc = ... in
  try_finalize process desc close desc
  ```

Les fonctions `read` et `single_write`



Conditions

- le descripteur doit être ouvert, le tampon de taille suffisante.

Valeur retournée

- le nombre d'octets effectivement lus ou écrits, au plus n.
- 0 signifie la fin de fichier pour les lectures.
- en général égal à n pour les écritures, sauf pour les entrée-sortie asynchrones (tuyau ou prise ouverte avec `O_NONBLOCK`). ou les écritures de très grande taille.

Lecture-écriture sur un descripteur

En OCaml

```
read : file_descr -> string -> int -> int -> int
single_write : file_descr -> string -> int -> int -> int
```

`single_write` effectuera systématiquement une écriture partielle pour des écritures de taille trop grandes (taille dépendant du système).

Lecture “forcée”

```
let rec really_read fd buffer start length =
  if length <= 0 then () else
    match read fd buffer start length with
      0 -> raise End_of_file
    | r -> really_read fd buffer (start + r) (length - r);;
```

Lecture-écriture sur un descripteur

En OCaml

```
read : file_descr -> string -> int -> int -> int
single_write : file_descr -> string -> int -> int -> int
```

`single_write` effectuera systématiquement une écriture partielle pour des écritures de taille trop grandes (taille dépendant du système).

Écriture “forcée”

La fonction `write` de même interface que `single_write` répète l’écriture jusqu’à écrire tout ce qui était demandé. Mais attention ! en cas d’erreur, elle reporte l’erreur mais ne dit pas combien d’octets ont été écrits, ce qui peut poser un problème pour l’écriture dans des tuyaux en présence d’interruptions.

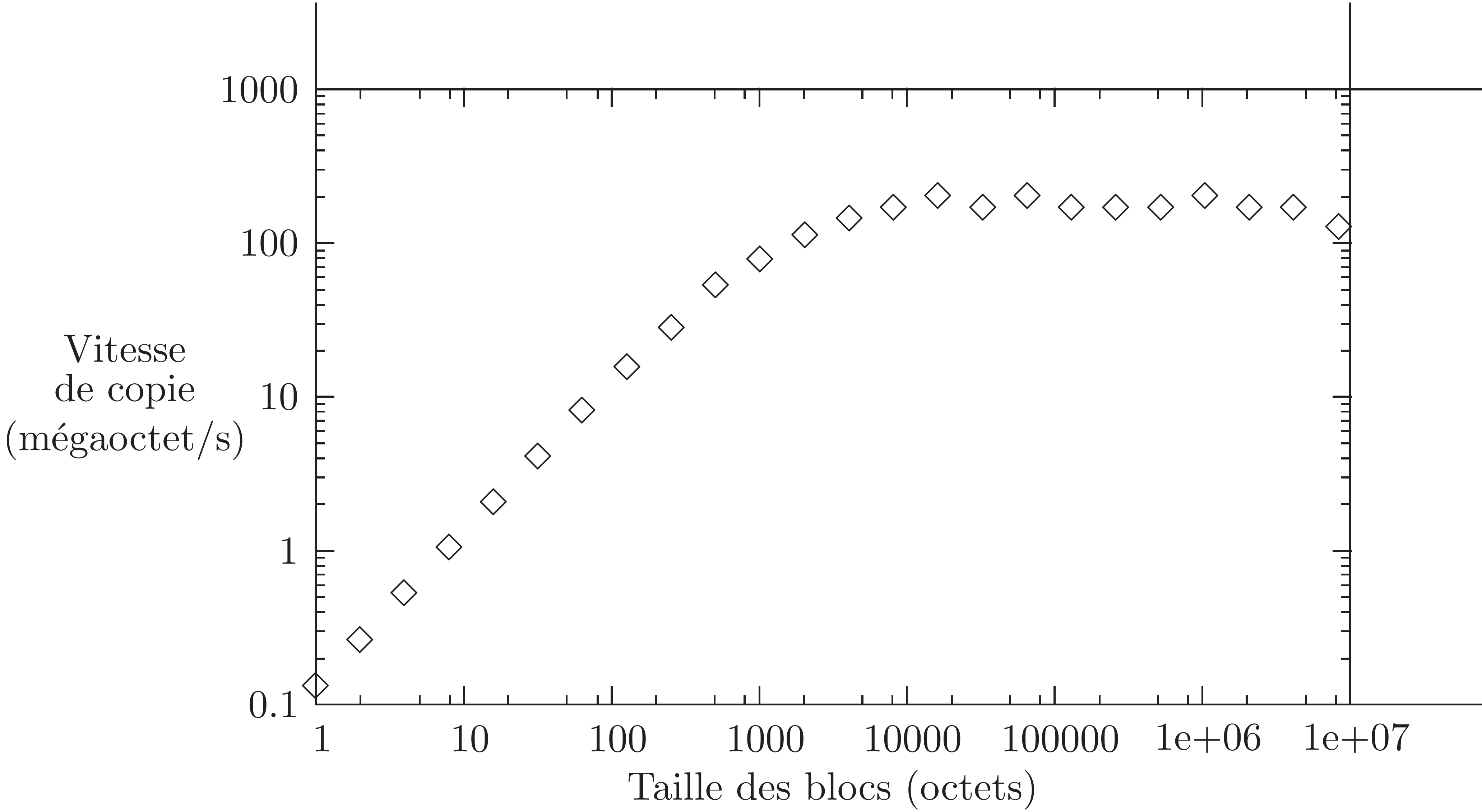
Exercice Réécrire `write` en utilisant `single_write`.

Copie de fichiers

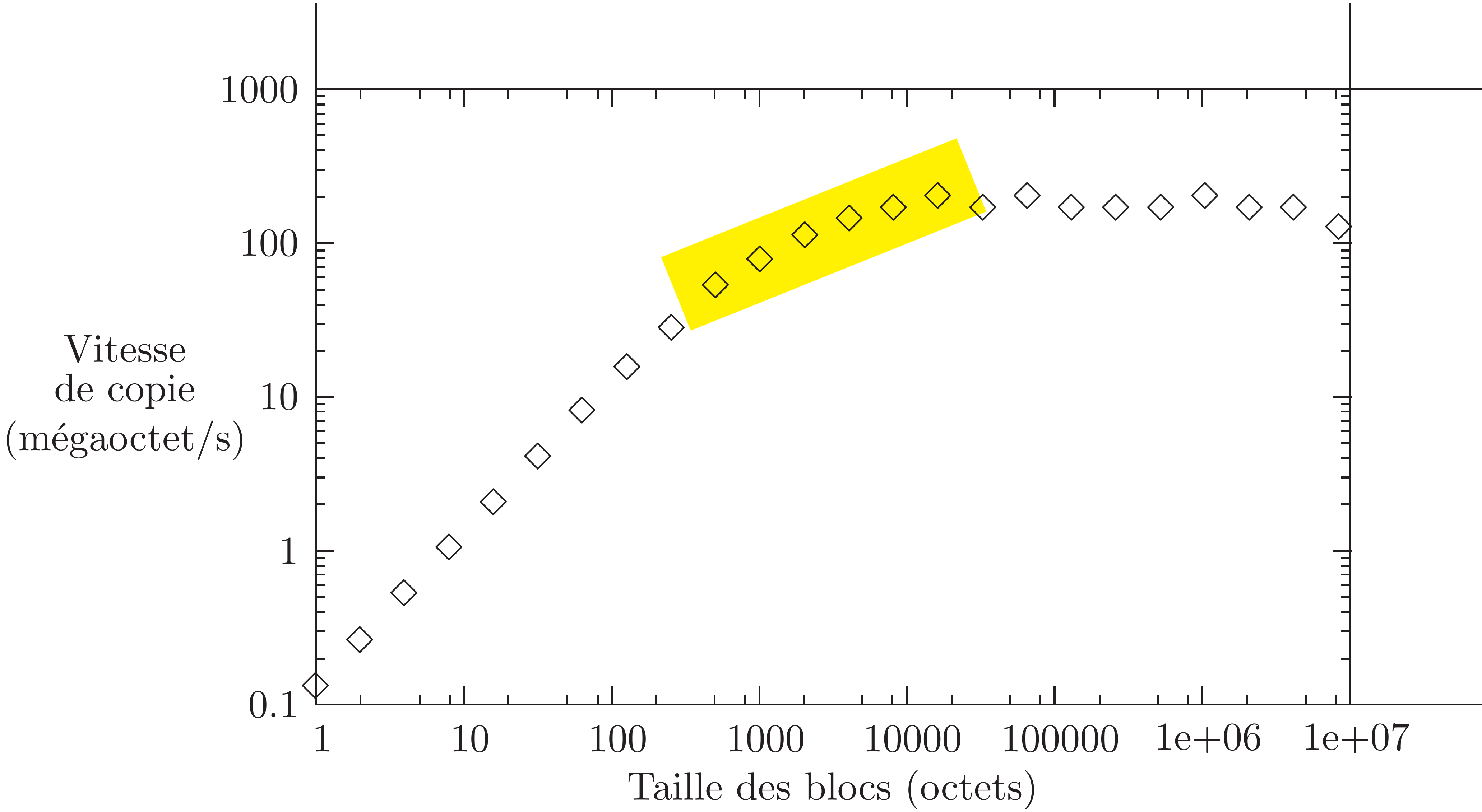

```
let buffer_size = 8192;;
let buffer = String.create buffer_size;;

let file_copy input_name output_name =
  let fd_in = openfile input_name [O_RDONLY] 0 in
    try openfile output_name [O_WRONLY;O_CREAT;O_TRUNC] 0o666
    with x -> close fd_in; raise x in
  let rec copy_loop () =
    match read fd_in buffer 0 buffer_size with
      0 -> ()
    | r -> ignore (write fd_out buffer 0 r); copy_loop () in
  try_finalize copy_loop ()
    (fun () -> close fd_out; close fd_in) ();;
```

Exercice : Donner une version protégée contre les erreurs.

Vitesse de copie/taille des blocs

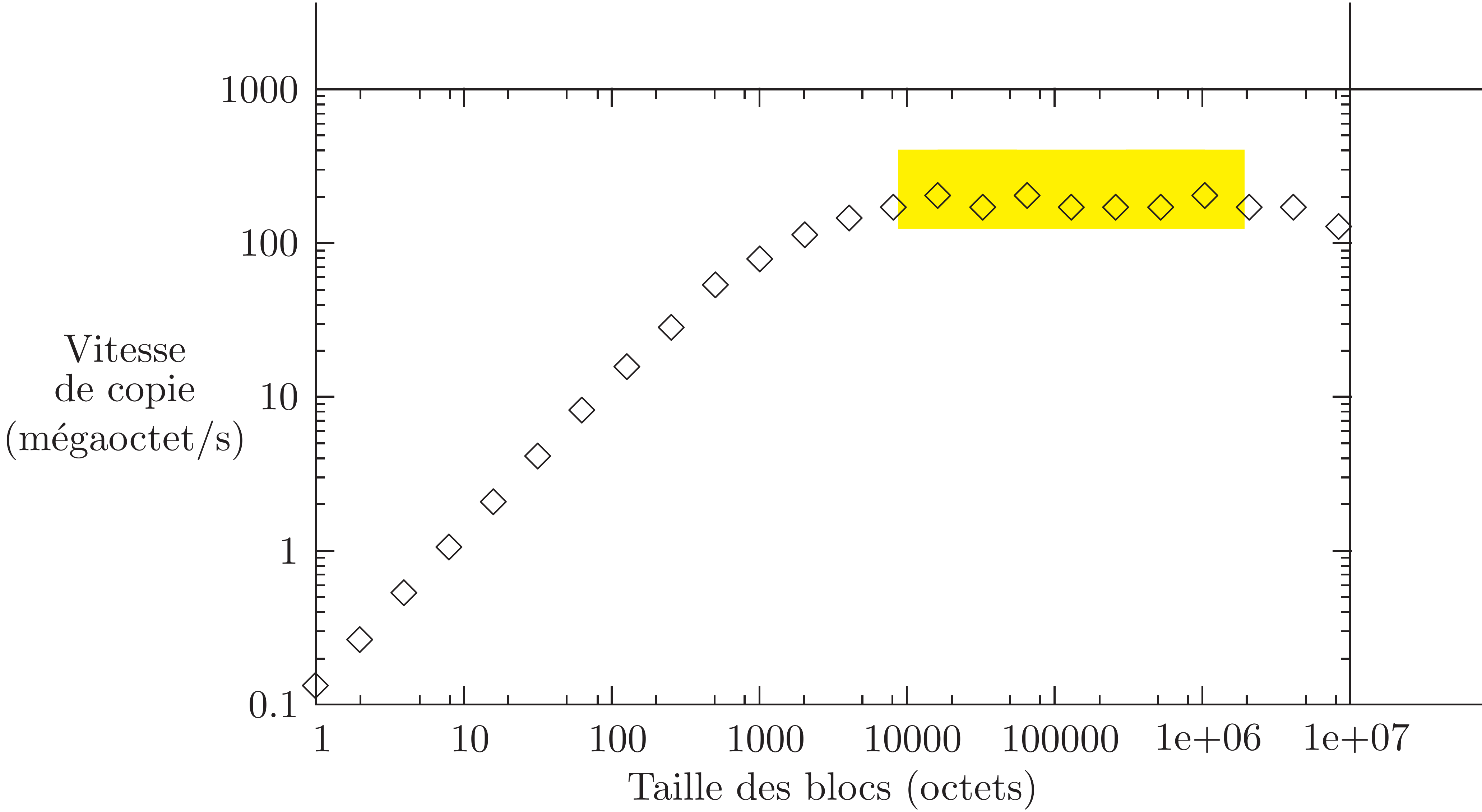


Vitesse de copie/taille des blocs



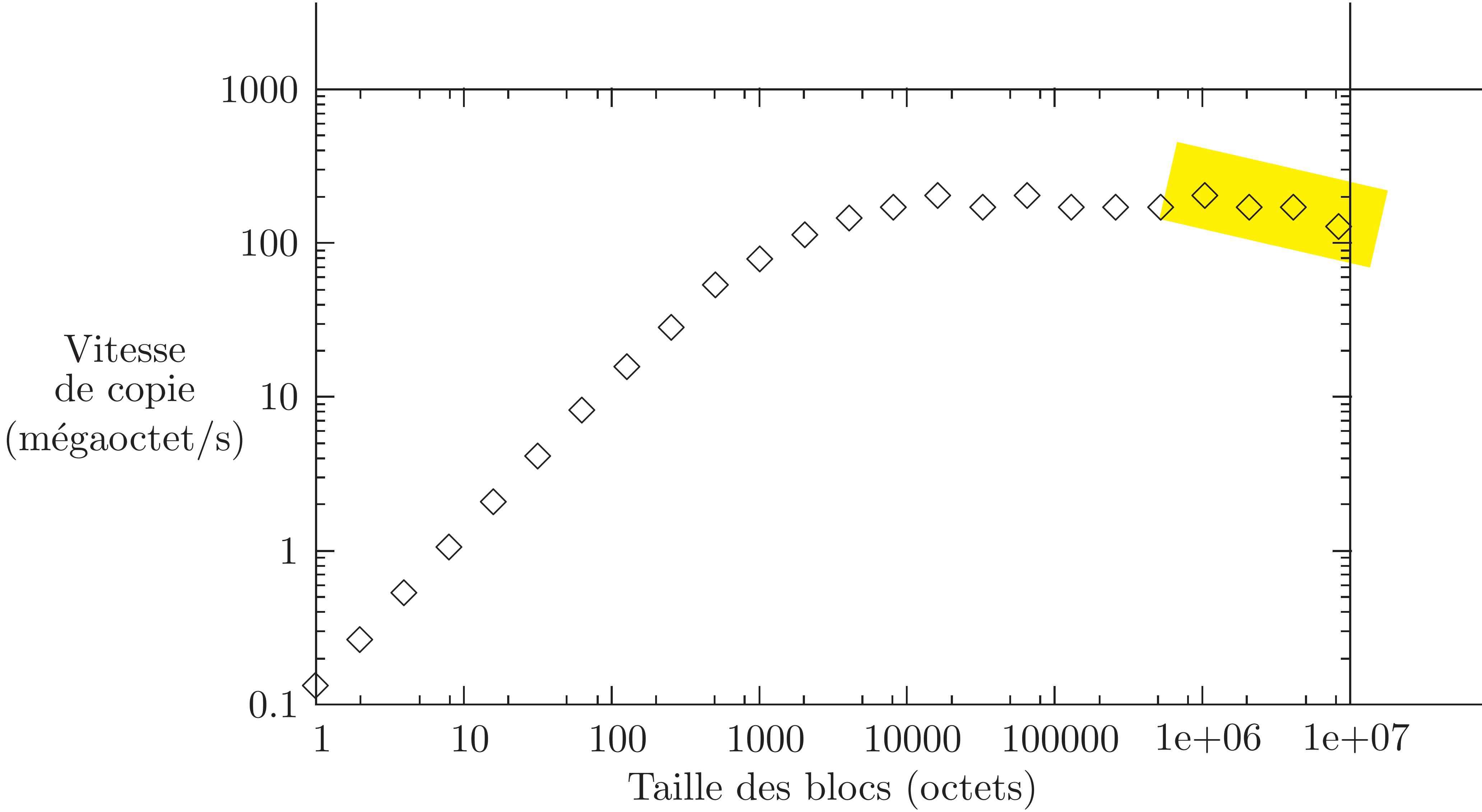
- **Blocs moyens ($0.5K \leq \cdot \leq 10K$) :** la copie est presque optimale (sur tous types de fichiers).

Vitesse de copie/taille des blocs



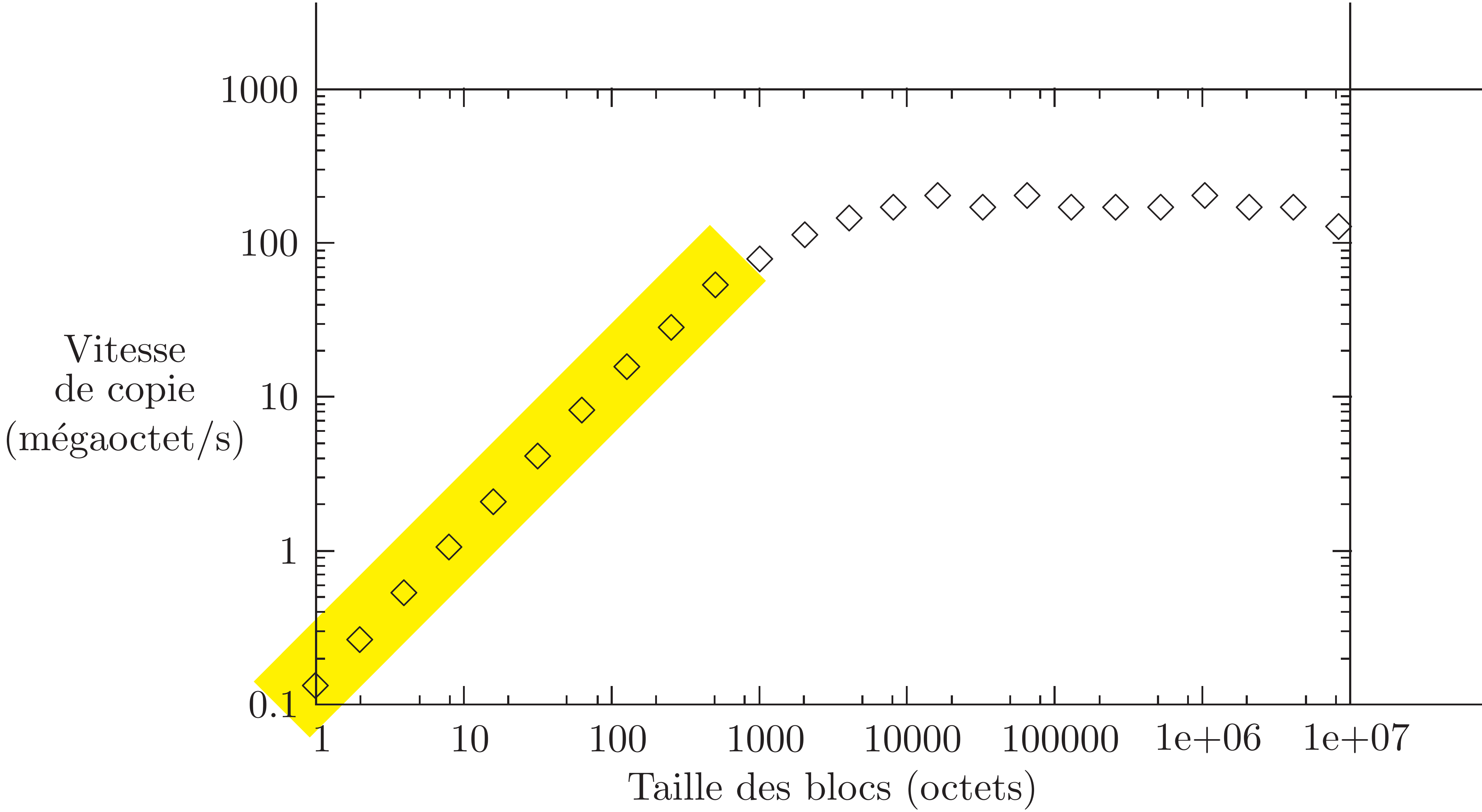
- **Gros blocs ($10K - 100K$) :** la copie est optimale et constante (sur des gros fichiers)

Vitesse de copie/taille des blocs



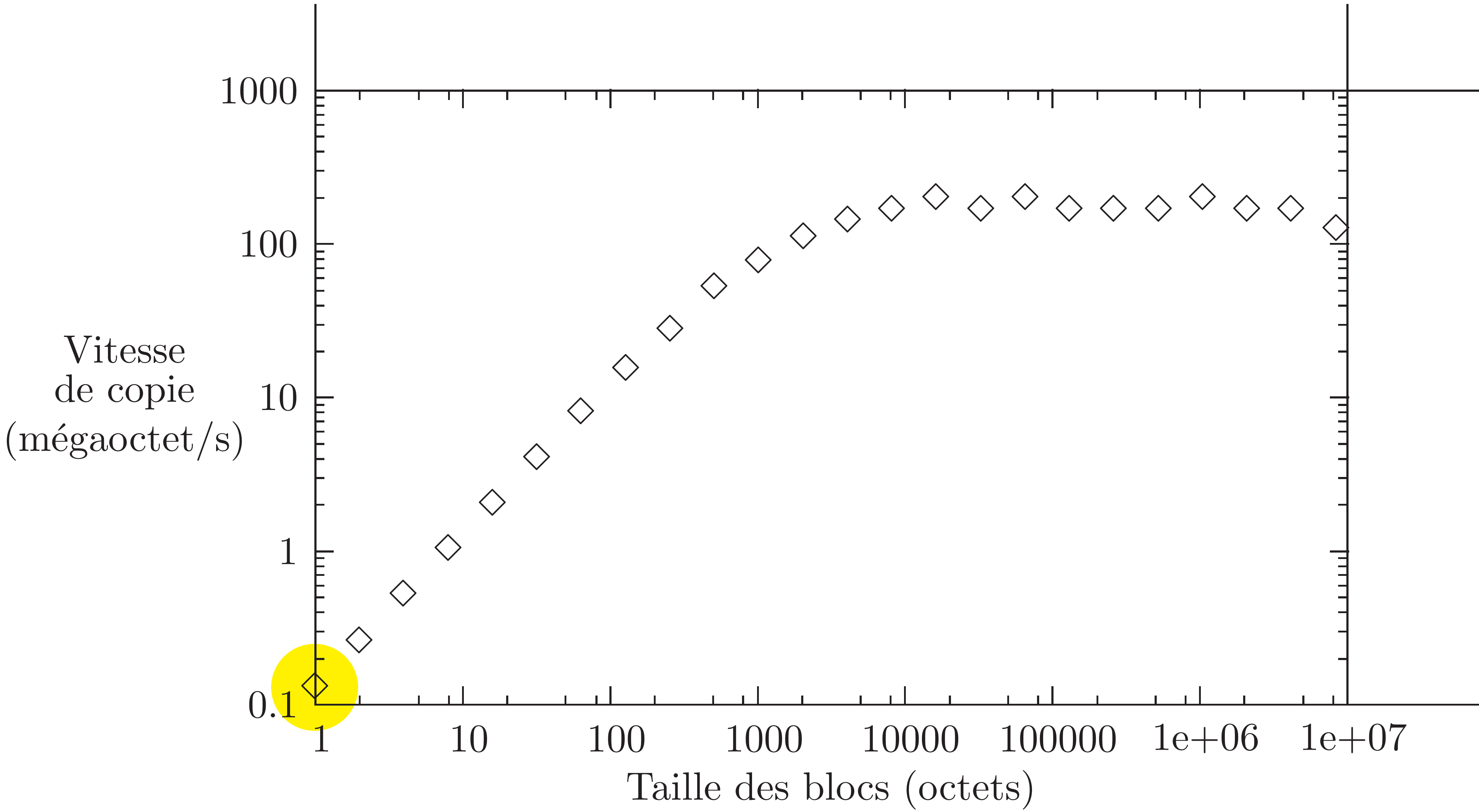
- **Très gros blocs ($\geq 100K$) :** la copie est légérement pénalisée par l'allocation.

Vitesse de copie/taille des blocs



- **Petits blocs ($< 0.5K$:** la copie est pénalisée par le nombre d'appels système.

Vitesse de copie/taille des blocs



- Coût d'un appel système = 1 à 10 micro-secondes.

Coût des appels systèmes

Très approximativement $\begin{cases} \text{1 à 10 micro-secondes} \\ \text{500 à 1000 instructions} \end{cases}$

C'est cher ! Il ne faut pas en abuser (*i.e.* en faire pour rien).

Mais ce n'est pas un drame sur les machines modernes... (*e.g.* `whizzytex`, demo ?) : faire ce qu'il y a à faire.

Pourquoi ? Ce n'est pas un simple appel de fonction...

- Le programme appelle une fonction de librairie ;
- Celle-ci prépare les arguments si besoin ;
- L'appel système est "interprété" : le programme passe le numéro de l'appel dans un registre et les arguments sur la pile.
- le programme exécute une instruction privilégiée "trap" et le calcul passe en mode privilégié.
- le système décode le numéro de l'appel, en déduit le nombre et la position des arguments, les recopie dans une zone réservée.
- le calcul est effectué... *et retourné avec un protocole inverse.*

Si le programme doit lire les octets par petits blocs ou un par un, c'est très inefficace de le faire par l'appel système `read`.

Lecture temporisée

- Lecture d'un gros bloc dans un tampon.
- Lecture du tampon petit à petit.

puis remplissage du tampon lorsqu'il est vide, etc.

Écriture temporisée (processus analogue) :

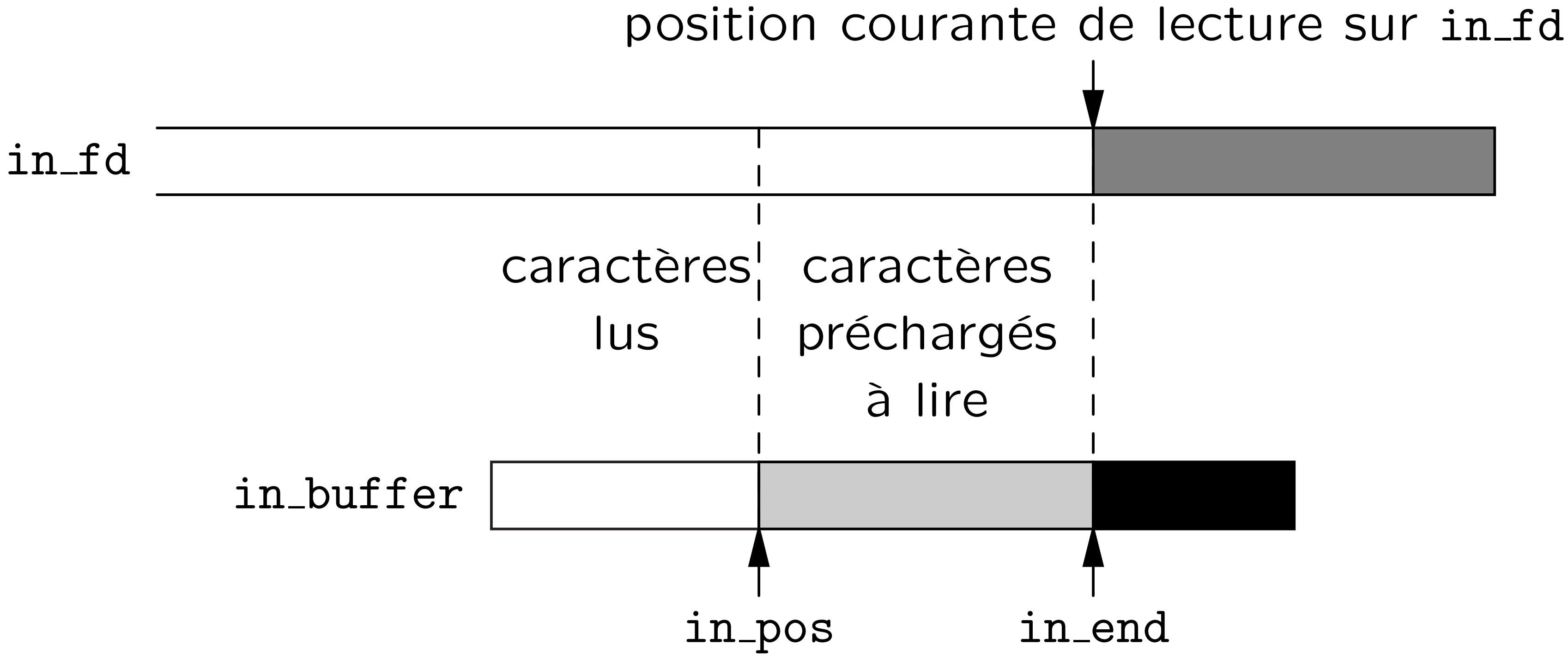
- Le tampon contient des écritures en retard
- On vide le tampon lorsqu'il est plein et avant sa fermeture.

Ces opérations sont fournies en librairie

Gain/perte des entrées-sorties temporisées ?

Gain important. Perte faible : au pire, une recopie en plus.

Lecture temporisée



Interface :

```
type in_channel
exception End_of_file
val open_in : string -> in_channel
val input_char : in_channel -> char
val close_in : in_channel -> unit

type out_channel
val open_out : string -> out_channel
val output_char : out_channel -> char -> unit
val close_out : out_channel -> unit
```

Les tampons


```
type in_channel =
  { in_buffer: string;
    in_fd: file_descr;
    mutable in_pos: int;
    mutable in_end: int };;
exception End_of_file

let buffer_size = 8192;;
let open_in filename =
  { in_buffer = String.create buffer_size;
    in_fd = openfile filename [O_RDONLY] 0;
    in_posa = 0;
    in_end = 0 };;

let close_in chan =  close chan.in_fd;;
```

```
let input_char chan =
  if chan.in_pos < chan.in_end then
    begin
      let c =  chan.in_buffer.[chan.in_pos] in
      chan.in_pos <- chan.in_pos + 1; c
    end
  else
    begin
      match read chan.in_fd chan.in_buffer 0 buffer_size
      with
        0 -> raise End_of_file
      | r -> chan.in_end <- r;
          chan.in_pos <- 1;
          chan.in_buffer.[0]
    end;;
```

Exercice

- Programmer les fonctions d'écriture.
- Ajouter des fonctions travaillant sur les entiers, les chaînes.

Et vous obtiendrez une librairie d'I/O temporisées...

Verrouillage du contenu d'un fichier

Si plusieurs processus peuvent modifier un même fichier,

- utiliser un fichier auxilliaire comme verrou.
- verouiller directement une partie ou la totalité du fichier.

```
lockf : file_descr -> lock_command -> int -> unit
```

`L_ULOCK`	Déverrouille
`L_LOCK`	Verrouille en écriture, bloque si la région est prise
`L_TLOCK`	Verrouille en écriture, échoue si le région est prise
`L_TEST`	Teste si le verrou est libre
`L_RLOCK`	comme `L_LOCK` mais en lecture
`L_TRLOCK`	comme `L_TLOCK` mais en lecture

- **Un verrouillage est coopératif** : un processus qui ne vérifie pas les verrous peut écraser la région. *Solution* ?
- Difficulté pratique (*e.g.* mail) : coexistence de 3 mécanismes de vérouillages : par un fichier `lock`, par `flock` ou par `lockf`.

Positionnement dans un fichier

Les fichiers réguliers sont dits en « accès direct ».

- On peut se positionner à un endroit arbitraire du fichier sans lire les octets intermédiaires.
- Permet d'accéder efficacement à des fichiers structurés.
- Typiquement les bases de données où un fichier d'index décrit la position de certaines entrées dans d'autres fichiers.
  (Autre exemple : la commande `tar` pour archiver les fichiers.)

En OCaml

```
lseek : file_descr -> int -> seek_command -> int
type seek_command = SEEK_SET | SEEK_CUR | SEEK_END
```

Retourne la nouvelle position courante.

Exemple : `lseek fd (-3) SEEK_END` se place 3 octets avant la fin du fichier pointé par `fd` et retourne la taille moins 3 octets.

La commande tail

La commande `tail` retourne les N dernières lignes d'un fichier.

Une implémentation naïve lit le fichier, retient les N dernières lignes dans un tampon circulaire qui est affiché à la fin.

Une implémentation plus efficace compte les lignes à partir de la fin du fichier.

- on se positionne à la fin du fichier ;
- on lit un gros bloc ;
- on compte les retours à la ligne à l'envers dans le gros bloc.
- s'il y en a assez on les affiche, sinon on relit le gros bloc précédent, etc.

Option `-f` (voir man) : à la fin du fichier le descripteur est gardé ouvert et on essaye de lire de nouvelles données à intervalle régulier pour (donner l'illusion de) les afficher au fur et à mesure.

Fichiers Spéciaux

Philosophie Unix

En Unix, la communication passe par des « descripteurs de fichiers » que ceux-ci soient matérialisés (fichiers, périphériques) ou volatiles (communication entre processus par des tuyaux ou prises).

- donne une interface minimale uniforme à la communication de données.
- communication implémentée différemment selon le média.
- pour certains média, il existe des opérations supplémentaires.

Les fichiers spéciaux

Traite le cas des communications avec les périphéques.

Typiquement, ces fichiers sont dans le répertoire `/dev`

Deux types de fichiers spéciaux

Les fichiers de type caractère

Ce sont des flux de caractères : on ne peut lire ou écrire les caractère que dans l'ordre : typiquement les terminaux, périphériques sons, imprimantes, etc.

Les fichiers de type bloc

Le support est rémanent ou tamporisé : on peut lire les caractères par blocs, voire à une certaine distance donnée de façon absolue ou relative à la position courante : typiquement les disques.

Exemples de fichiers spéciaux

/dev/null

C'est le trou noir : il n'y a rien dedans et tout ce qu'on y met part à la poubelle. Utile pour ignorer les résultats d'un processus : on redirige la sortie vers `/dev/null` (voir cours suivants).

/dev/tty*

Ce sont les terminaux de contrôle.

/dev/pty*

Pseudo terminaux de contrôle : ils simulent un terminal (même interface), mais n'en sont pas.

/dev/hd*

Ce sont les disques.

Terminaux

En Unix, les opérations spécifiques aux fichiers spéciaux sont regroupées dans la commande `ioctl`.

Cette commande est un fourre-tout : son format, sa sémantique dépendent du type de periphériques du fichier sur lequel le descripteur est ouvert.

Elle n'est pas relevée en `OCaml`, sauf pour accéder aux paramètres de contrôle des terminaux :

```
tcgetattr : file_descr -> terminal_io
tcsetattr : file_descr -> setattr_when -> terminal_io -> unit
```

(Voir le polycopié.)

Exemple : lecture d'un mot de passe


```
let read_passwd message =
  let default = tcgetattr stdin in
  let silent =
    { default with
      c_echo = false; c_echoe = false;
      c_echok = false; c_echonl = false;} in
  print_string message;
  flush Pervasives.stdout;
  tcsetattr stdin TCSANOW silent;
  try_finalize input_line Pervasives.stdin
    (tcsetattr stdin TCSANOW) default;;
```

La commande tar (tape archive)

Description

Une archive (au format `tar`) est un fichier qui décrit toute une hiérarchie de fichiers.

Elle permettent de sauver une hiérarchie sur une bande, l'envoyer au travers du réseau, *etc.* pour la reconstituer à distance, ultérieurement ou sur un autre support.

Le format est portable d'un système d'exploitation à un autre.

C'est un exemple de système de fichiers simplifié.

Description

Une archive (au format `tar`) est un fichier qui décrit toute une hiérarchie de fichiers.

Utilisation

La commande `tar` appelée avec les options `-cf` :

`tar -cf` a f_1 f_2 ... f_n

crée une archive à partir d'un ensemble des fichiers (ordinaires, liens durs ou symboliques ou répertoires).

L'archive est constituée d'un seul fichier.

La hiérarchie peut être recréée à partir de l'archive par `tar` avec les options `-xf` :

`tar -cf` a f_1 f_2 ... f_n

Une `tar` archive

- est une suite d'enregistrements, un par fichier.
- terminée par la fin de fichier ou un bloc vide, éventuellement suivi de blocs vides.
- comportant au moins 20 blocs (10240 octets).

Un enregistrement

- comporte une entête codée sur un bloc (512 octets).
- le contenu du fichier sur un multiple de blocs.

Types des fichiers

'\0' ou '0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'
REG	LINK	LNK	CHR	BLK	DIR	FIFO	CONT

Offset	Length	Codage	Nom	Description
0	100	chaîne	`name`	Nom du fichier
100	8	octal	`perm`	Mode du fichier
108	8	octal	`uid`	ID de l'utilisateur
116	8	octal	`gid`	ID du groupe de l'utilisateur
124	12	octal	`size`	Taille du fichier3
136	12	octal	`mtime`	Date de la dernière modification
148	8	octal	`checksum`	Checksum de l'entête
156	1	caractère	`kind`	Type de fichier
157	100	octal	*link*	Lien
257	8	chaîne	`magic`	Signature (`"ustar\032\032\0"`)
265	32	chaîne	`user`	Nom de l'utilisateur
297	32	chaîne	`group`	Nom du groupe de l'utilisateur
329	8	octal	*major*	Ident. majeur du périphérique
337	8	octal	*minor*	Ident. mineur du périphérique
345	167			Remplissage

Analyse de la représentation

Résumé

- Les fichiers sont contigüs
- La hiérarchie n'est pas modifiable (retrait)
- On peut seulement étendre la hiérarchie à la fin de l'archive.
- Toutes les opérations ne sont pas permises (ex. liens durs).

Avantages

- Simplicité
- Portabilité
- Écriture incrémentale, sans retour en arrière

Invonvénients

- Hérarchie non modifiable, pas de recyclage.
- Déplacement dans la hiérarchie difficile sans lecture d'une grande partie de l'archive.

Problème avec la représentation contigües

...	Fichier A	Libre	Grand Fichier B	Fich. C	...

- On ne peut pas toujours agrandir un fichier existant (*e.g.* B)
- Fragmentation : il peut ne plus y avoir de place pour ranger un fichier de grande taille alors qu'il y a plein de petits trous.
- Problème des défauts matériels (zône défectueuse).

Représentation par blocs

- Le contenu d'un fichier est une suite non contigües de blocs de taille fixe
- Un fichier est identifié par un inode qui comporte des méta-informations et identifie la suite des blocs le composant.

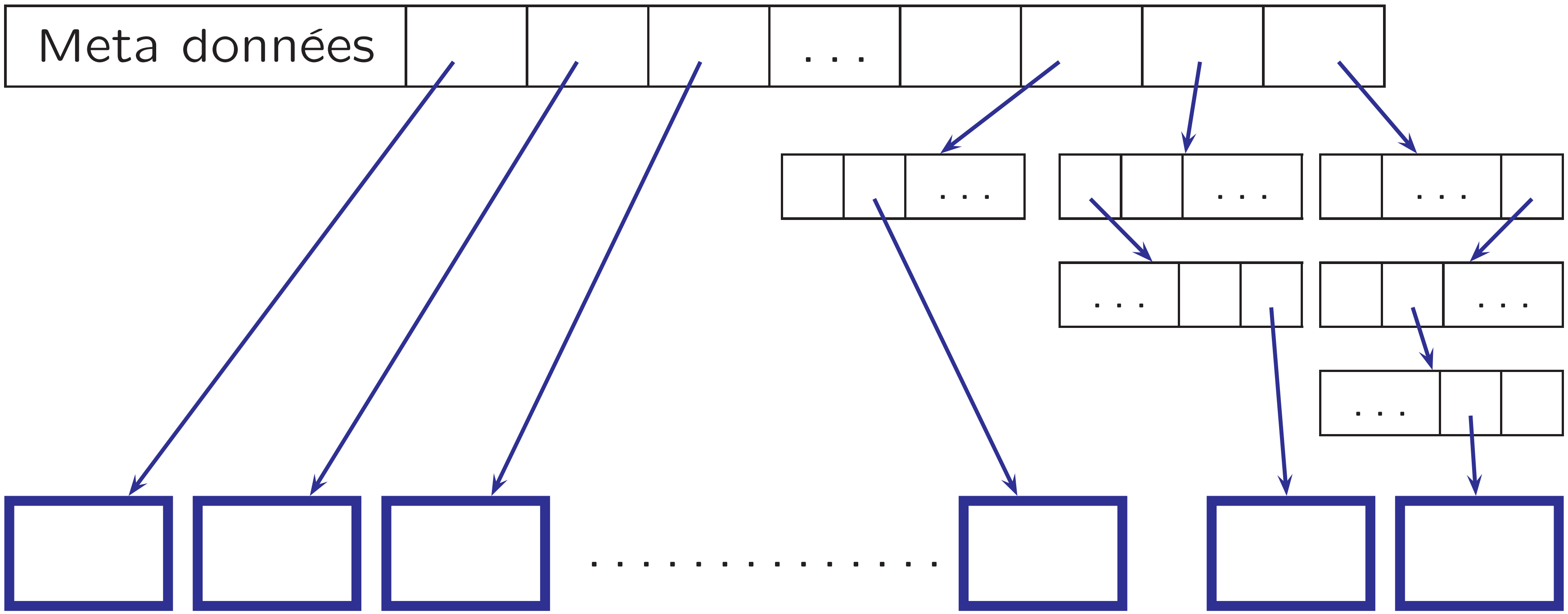
La structure des fichiers sur le disque

Disque peut contenir plusieurs systèmes de fichier.

Système de fichiers (partition) est une séquence de blocs avec la structure logique suivante :

- Secteur de boot : permet éventuellement de démarrer sur le disque, il peut être vide.
- Super bloc : contient l'état du disque : nombre de fichiers (inodes), place libre, où trouver de l'espace libre.
- Liste des inodes : leur nombre est déterminé au formattage.
- Blocs de donnée : un bloc appartient au plus à un fichier.
  - Tous les blocs ont la même taille (régularité = simplicité).
  - Une plus grande taille permet (dans une certaine mesure) un transfert de données plus économique, mais augmente en moyenne la place perdue sur le disque.

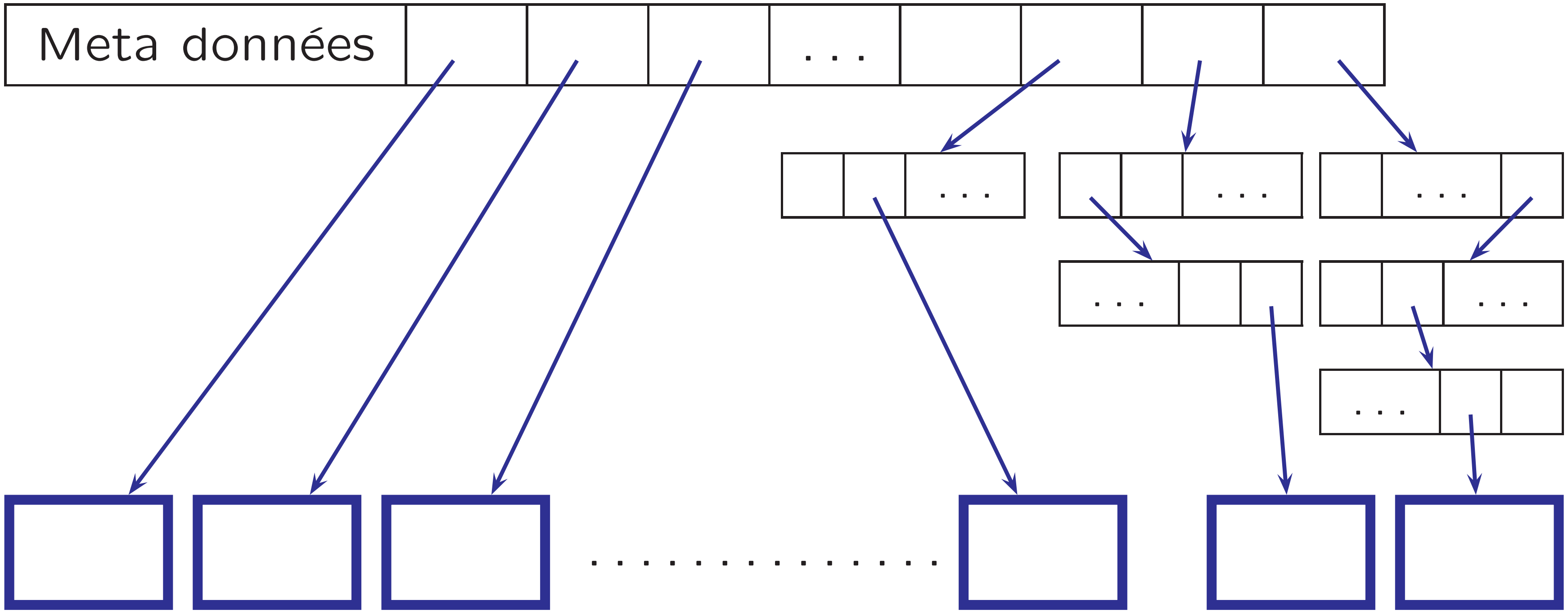
Structure d'un inode



Structure

- Meta-données (taille, dates, *etc.*)
- Liste des k-premiers blocs (sans indirection)
- Blocs de simple, double et tripple indirection.

Structure d'un inode



Avantages

- Efficacité : accès direct pour les petits fichiers
- Capacité : 4 GO si blocs de 1 KO ou 4 TO si blocs de 4 KO. (numéro d'inode codé sur un entier 32 bits donc 4 O)
- Bloc numéro 0 inexistant, réprésente un bloc de '\000', alloué si besoin (écriture d'un octet non nul).

Représentation des répertoires

Même représentation que les fichiers

- Le type de fichier est différent dans les méta-données.
- Le contenu du fichier est une suite de paires formées d'un numéro-d'inode et d'un chemin.

Avantages

- Le système utilise le même code pour lire le contenu d'un bloc sur le disque, qu'il s'agisse d'un répertoire ou d'un fichier.
  La différence est son interprétation une fois lu en mémoire.
- Il n'y a pas de limite a priori au nombre d'entrées par répertoire.

Gestion des blocs libres

Tous les blocs libres sont chaînés dans une liste

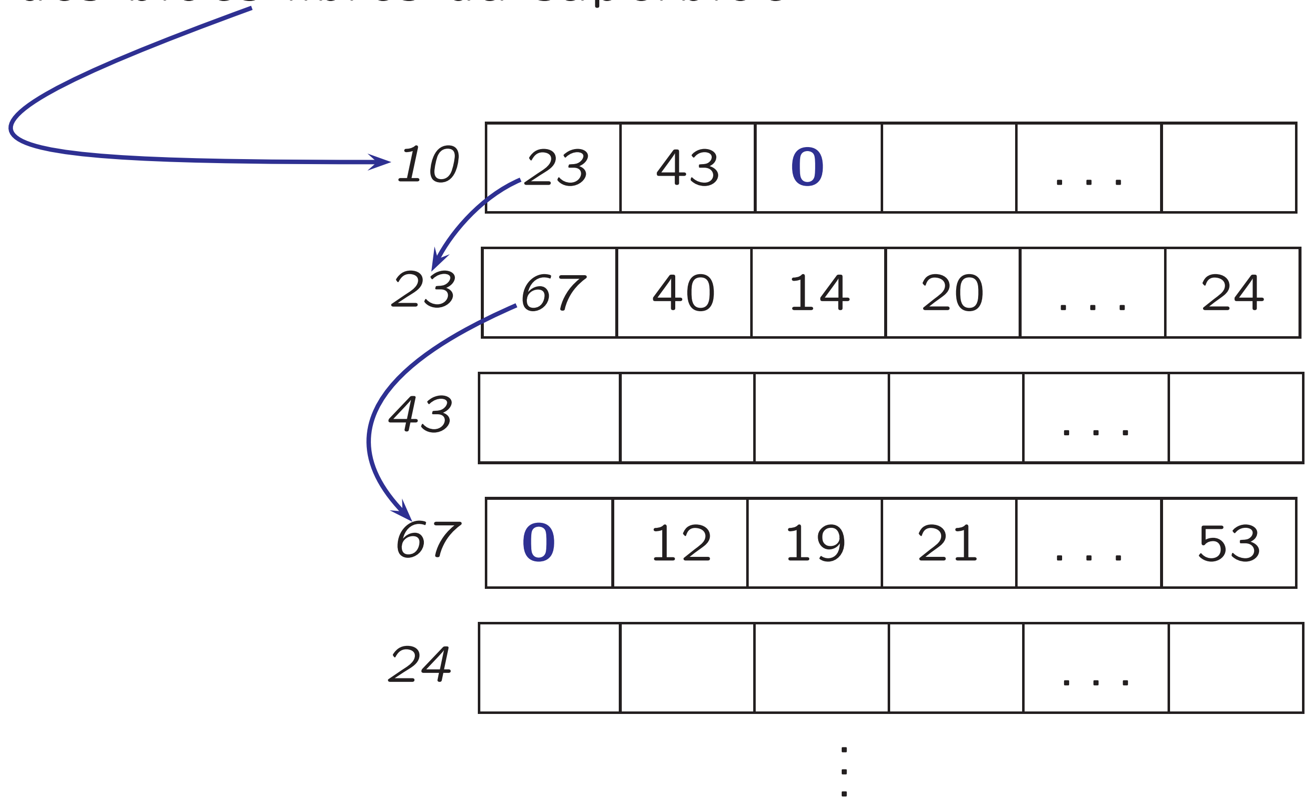
- Construite au formattage du disque.
- Lorsque qu'un inode est libéré, ses blocs sont libérés et sont remis dans la liste des blocs libres.

Organisation de la liste libre

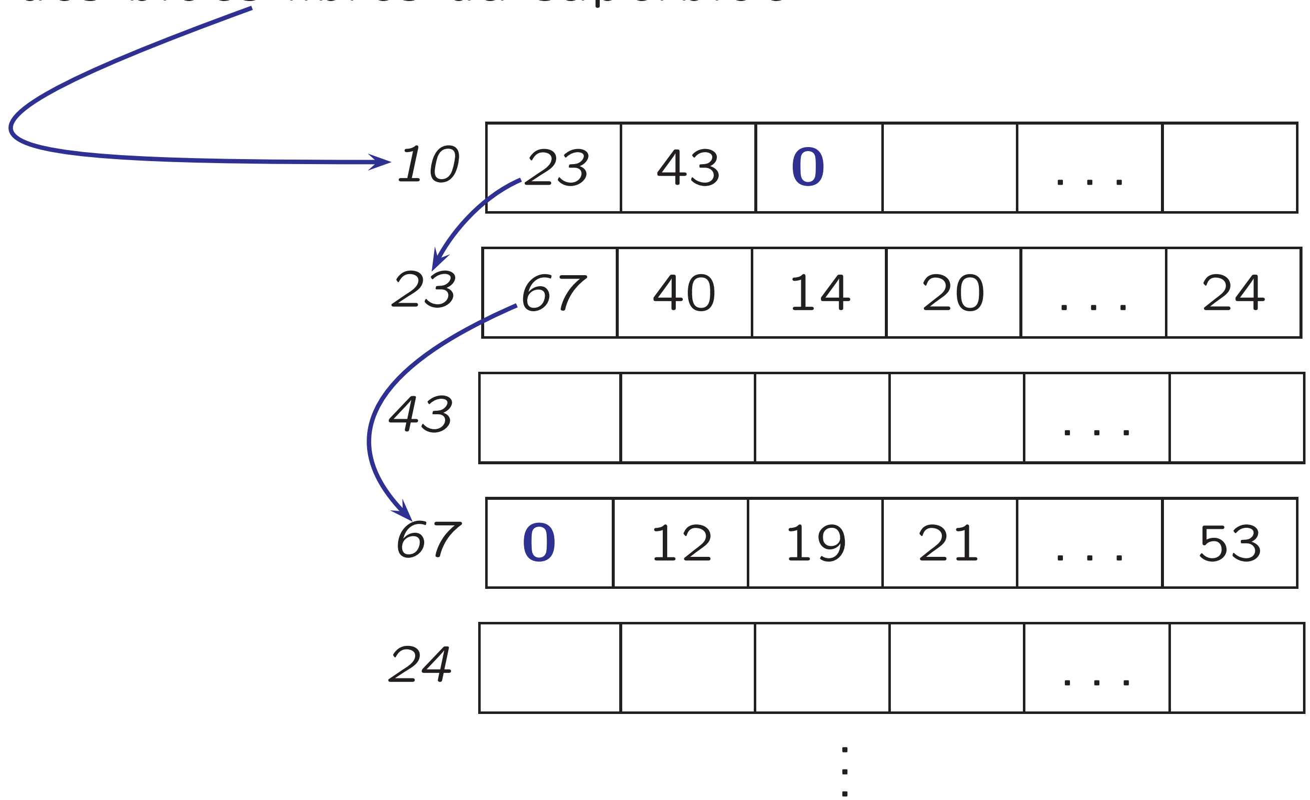
- Un champ du super bloc pointe vers la liste des blocs vides.
- On utilise des blocs libres pour former les cellules de la liste des blocs libres.
  - Leur contenu est une liste de numéros de blocs libres.
  - Le premier d'entre eux est la cellule suivante de la liste.

Schématiquement

Liste des blocs libres du superbloc



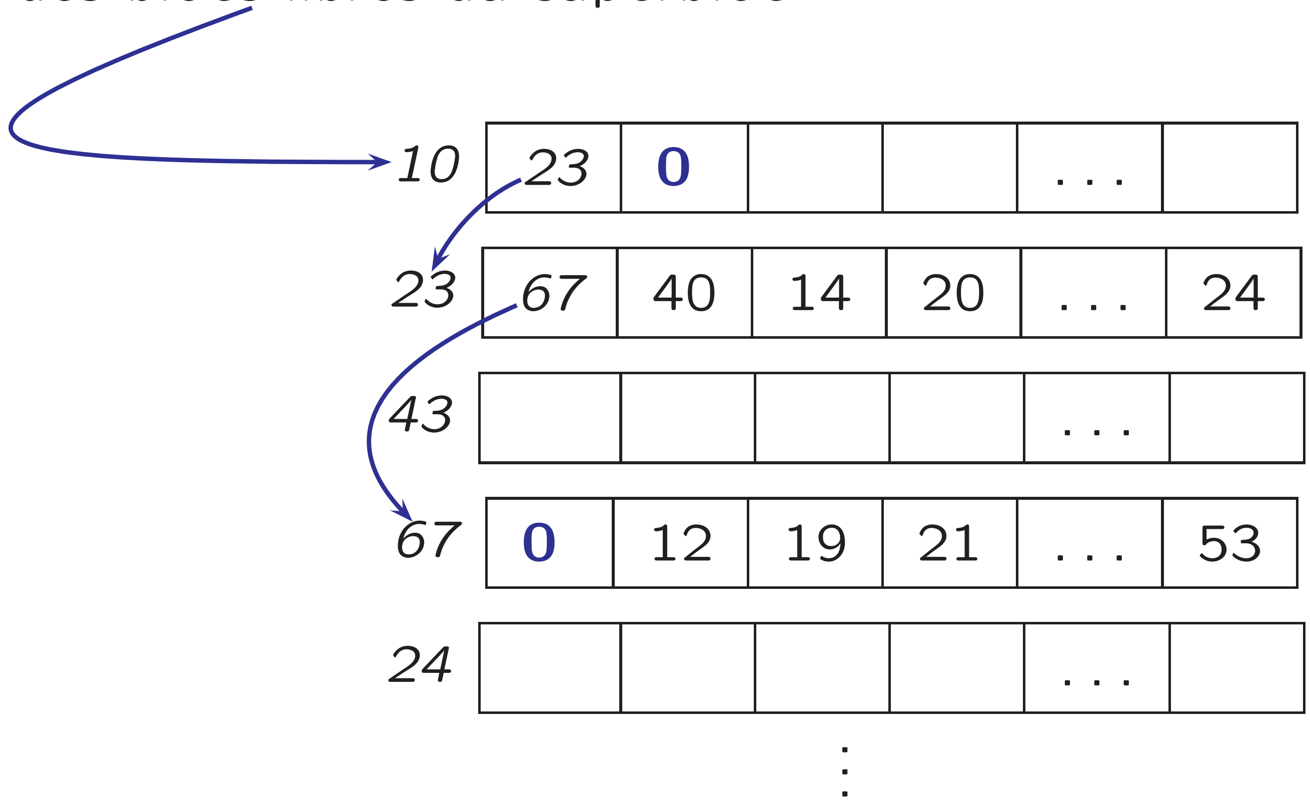
Liste des blocs libres du superbloc



- Allocation d'un bloc

Schématiquement

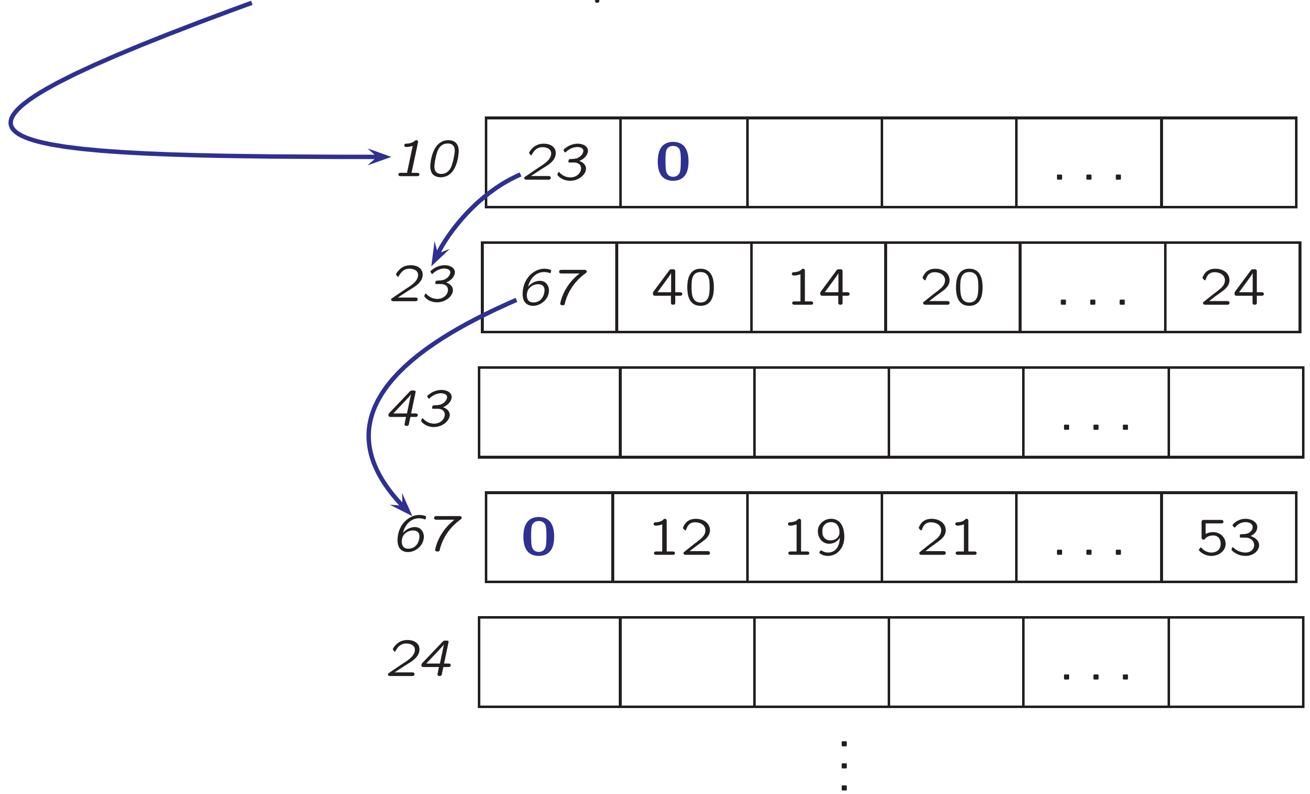
Liste des blocs libres du superbloc



- Allocation d'un bloc $\longrightarrow$ 43

Schématiquement

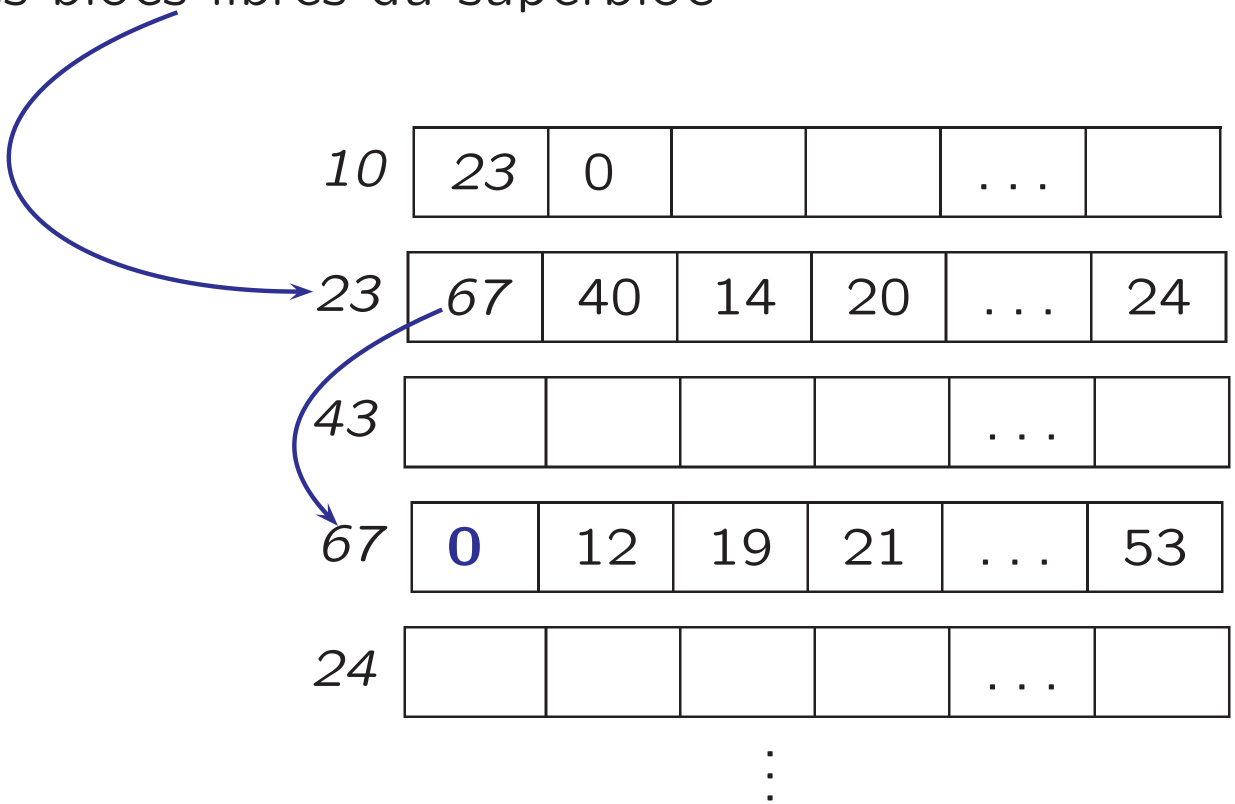
Liste des blocs libres du superbloc



- Allocation d’un bloc $\longrightarrow$ 43
- Allocation d’un bloc

Schématiquement

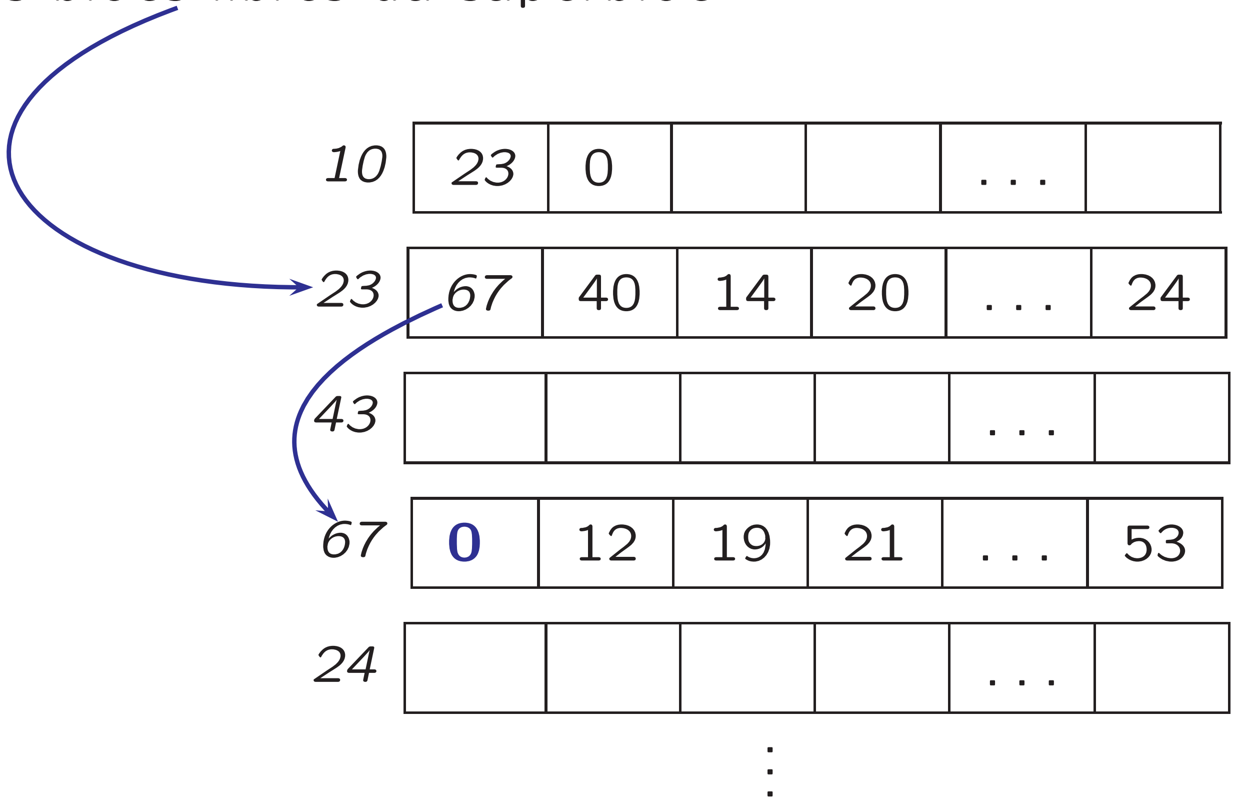
Liste des blocs libres du superbloc



- Allocation d'un bloc $\longrightarrow$ 43
- Allocation d'un bloc $\longrightarrow$ 10

Schématiquement

Liste des blocs libres du superbloc



- Allocation d'un bloc $\longrightarrow$ 43
- Allocation d'un bloc $\longrightarrow$ 10

Une solution mixte

- Liste partielle des inodes libres.
- Numéro du plus grand inode dans la liste libre.
- Nombre total d'inodes libres.

Recherche d'un inode libre

- On les prend dans la liste des inodes libres.
- Quand il n'y en a plus on reconstruit une liste partielle d'inodes libres en parcourant la liste des inodes à partir de l'ancien plus grand inode libre dans la liste libre (maintenant vide).

Explication des différences

- On peut tester si un inode est libre par son nombre de liens.
- Ce n'est pas possible pour les blocs sans un parcours du disque complet : un bloc n'a pas de structure.
- Les blocs sont beaucoup plus sollicités que les inodes.