Persistence : cohérence du système de fichiers et journalisation

Thibaud Martinez
thibaud.martinez@dauphine.psl.eu

Dauphine | PSL
UNIVERSITÉ PARIS

Cette présentation couvre le chapitre 42 de Operating Systems: Three Easy Pieces.

Les diapositives sont librement adaptées de diapositives de *Youjip Won (Hanyang University)*.

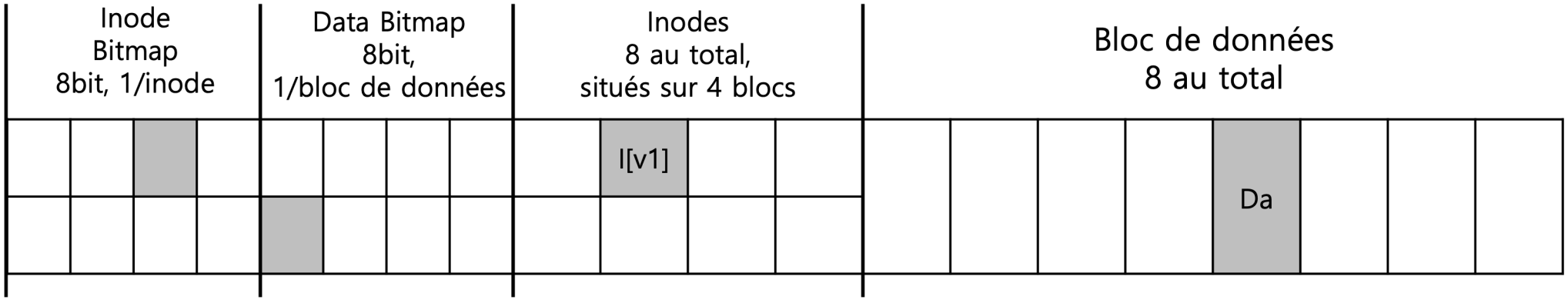
Cohérence après une défaillance

- Les structures de données des systèmes de fichiers doivent **persister**. Elle sont stockées sur des dispositifs qui conservent les données à long terme malgré les **coupures de courant**.
- L'un des principaux défis auxquels est confronté un système de fichiers est la **mise à jour des structures de données persistantes en cas de panne** de courant ou de défaillance du système.

Problème de cohérence : exemple

- Ajout d'un seul bloc de données (4 KB) à un fichier existant.
- `open()` → `lseek()` → `write()` → `close()`

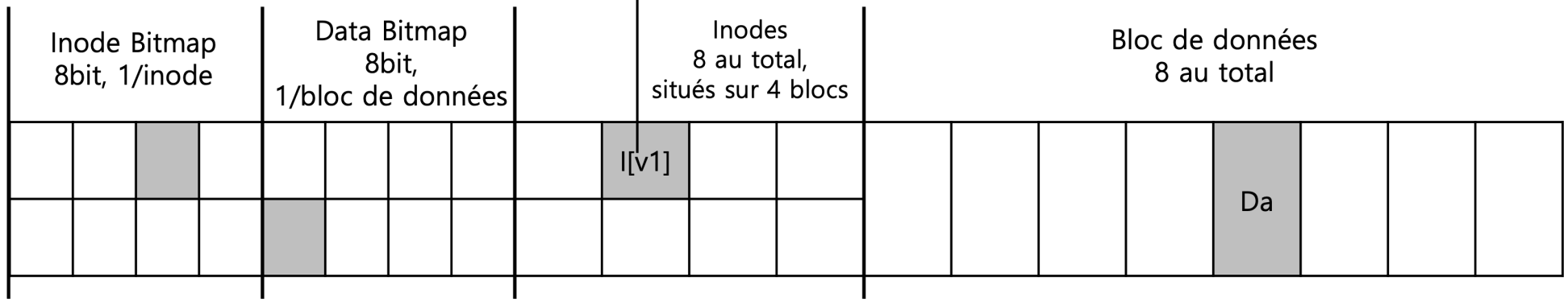
État avant l'ajout



- Un seul inode est alloué (inode numéro 2).
- Un seul bloc de données est alloué (bloc de données 4).
- L'inode est noté `I[v1]`.

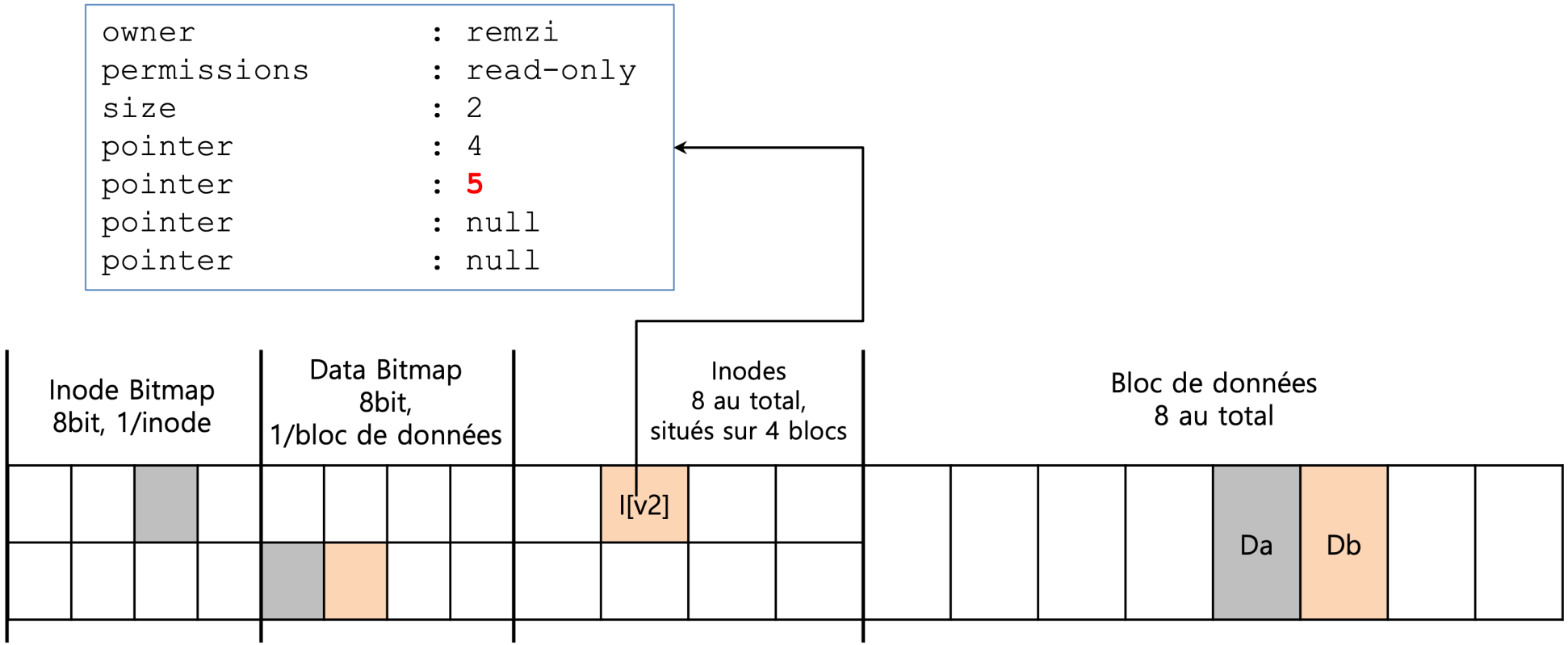
État de l'inode `I[v1]` (avant l'ajout)

```
owner           : remzi
permissions     : read-only
size            : 1
pointer         : 4
pointer         : null
pointer         : null
pointer         : null
```



- La taille du fichier est de 1 (un bloc alloué).
- Le premier pointeur direct pointe vers le bloc 4 (`Da`).
- Les 3 autres pointeurs directs sont `null` (non utilisés).

Après l'ajout du bloc de données



- Mise à jour du *data bitmap*.
- Mise à jour de l'inode (`I[v2]`)
- Un nouveau bloc de données est alloué (`Db`)

Pour réaliser la transition, le système effectue **3 écritures** distinctes sur le disque :

- inode `I[v2]` ;
- bitmap de données `B[v2]` ;
- bloc de données `Db` .

Ces écritures ne se produisent généralement pas immédiatement : l'inode mis à jour, le bitmap et les nouvelles données peuvent rester dans la mémoire tampon.

Si un crash survient après qu'une ou deux de ces écritures aient eu lieu, mais pas les trois, le système de fichiers risque de se retrouver dans un **état incohérent**.

Scénarios de défaillance

Imaginons qu'une seule écriture réussisse, il y a 3 résultats possibles :

1. **Seul le bloc de données (`Db`) est écrit sur le disque**
    - Les données sont sur le disque, mais il n'y a pas d'inode.
    - C'est donc comme si l'écriture n'avait jamais eu lieu.

    ✅ *Ce cas ne pose aucun problème.*

2. **Seul l'inode mis à jour (`I[v2]`) est écrit sur le disque**
    - L'inode pointe vers l'adresse disque `5`.
    - Mais `Db` n'a pas encore été écrite à cet endroit.
    - Nous lirons des données inutiles (ancien contenu de l'adresse `5`) sur le disque.

    💥 *Incohérence du système de fichiers.*

3. **Le bitmap mis à jour (`B[v2]`) est écrit sur le disque.**
    - Le bitmap indique que le bloc `5` est alloué.
    - Mais il n'y a pas d'inode qui pointe vers lui.
    - Ainsi, le système de fichiers est à nouveau incohérent.

💥 *Fuite d'espace, le bloc 5 ne sera jamais utilisé par le système de fichiers.*

Imaginons que deux écritures réussissent et que la dernière échoue.

1. **L'inode(`I[v2]`) et le bitmap(`B[v2]`) sont écrits sur le disque, mais pas les données (`Db`)**
    - Les métadonnées du système de fichiers sont parfaitement cohérentes.

    💥 *Le bloc `5` contient des données incorrectes.*
2. **L'inode(`I[v2]`) et le bloc de données (`Db`) sont écrits, mais pas le bitmap (`B[v2]`)**
    - L'inode pointe vers les données correctes sur le disque.

    💥 *Incohérence entre l'inode et l'ancienne version du bitmap (`B[v1]`).*

3. **Le bitmap (`B[v2]`) et le bloc de données (`Db`) sont écrits, mais pas l'inode (`I[v2]`)**
    - Incohérence entre l'inode et le bitmap de données.

    💥 Nous n'avons aucune idée du fichier auquel le bloc de données appartient.

Comment garantir un état cohérent en cas de défaillance ?

L'idéal serait de déplacer le système de fichiers d'un état cohérent à un autre de manière atomique.

Malheureusement, nous ne pouvons pas le faire facilement :

- Le disque n'effectue qu'une écriture à la fois.
- Des pannes ou des coupures de courant peuvent survenir entre ces mises à jour.

Deux approches pour résoudre le problème de la cohérence en cas de défaillance :

- **fsck** (*file system checker*)
- **journalisation** (*journaling* / *write-ahead logging*)

Le file system checker (fsck)

- Un outil permettant de **détecter les incohérences du système de fichiers et de les réparer**.
- fsck vérifie le super bloc, les tableaux de bits, l'état des inodes, les liens des inodes, etc.
- Une telle approche ne peut pas résoudre tous les problèmes.
  Exemple : le système de fichiers peut sembler cohérent mais un inode pointe vers des données incorrectes.
- Le seul véritable objectif est de s'assurer que les métadonnées du système de fichiers sont cohérentes en interne.

Fonctionnement de fsck

Super-bloc

- Vérifie d'abord que le super-bloc semble correct.
- Vérifie que la taille du système de fichiers est supérieure au nombre de blocs alloués.
- Si le super-bloc est suspect, le système peut décider d'utiliser une autre copie du super-bloc.

Blocs libres

- Analyse les inodes, les blocs indirects, etc., pour comprendre quels blocs sont actuellement alloués dans le système de fichiers.
- S'il existe une incohérence entre les tableaux de bits et les inodes, elle est résolue en se fiant aux informations contenues dans les inodes.

État de l'inode

- Chaque inode est vérifié pour corruption ou autres problèmes.
  Exemple : vérification du type (fichier normal, répertoire, lien symbolique, etc.)
- S'il y a des problèmes avec les champs de l'inode qui ne sont pas facilement corrigés, l'inode est considéré comme suspect et effacé.

Mauvais pointeurs de blocs

- Un pointeur est considéré comme "mauvais" s'il pointe vers quelque chose qui n'est pas dans sa plage de validité.
  Exemple : il a une adresse qui fait référence à un numéro de bloc plus grand que la taille de la partition.
- Dans ce cas, fsck ne peut rien faire de très intelligent ; il supprime simplement le pointeur.

Répertoires

- fsck ne comprend pas le contenu des fichiers utilisateur. Les répertoires contiennent des informations spécifiquement créées par le système de fichiers.
- Ainsi, fsck effectue des contrôles d'intégrité supplémentaires sur le contenu de chaque répertoire.

  Exemple :

  - S'assurer que `.` et `..` sont les premières entrées.
  - Chaque inode auquel il est fait référence dans une entrée de répertoire est-il alloué ?
  - Sassurer qu'aucun répertoire n'est lié plus d'une fois dans l'ensemble de la hiérarchie.

Limites de l'approche fsck

- Nécessite une connaissance approfondie du système de fichiers.
- Trop lent : l'analyse d'un disque entier peut prendre plusieurs minutes ou plusieurs heures.
- Avec l'augmentation de la capacité des disques, les performances du fsck sont devenues prohibitives.

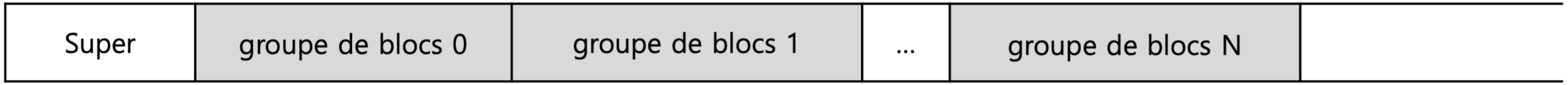
Journalisation (*write-ahead logging*)

- Lors de la mise à jour du disque, avant d'écraser les structures en place, écrivez d'abord une petite note décrivant ce que vous êtes sur le point de faire.
- L'écriture de cette note est la partie *"write ahead"*, et nous l'écrivons dans une structure que nous organisons comme un journal (*log*).
- En écrivant la note sur le disque, on garantit que si un crash se produit pendant la mise à jour des structures mises à jour, on peut revenir en arrière et lire la note et réessayer.
- Ainsi, on sait exactement ce qu'il faut réparer après un crash, au lieu de devoir parcourir tout le disque.

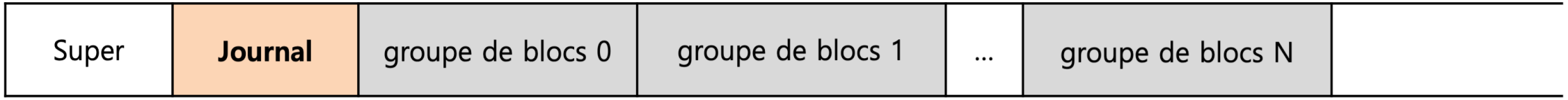
Journalisation sur les systèmes de fichiers ext3

- La plupart des structures du système sont identiques à celles de Linux ext2.
- La nouvelle structure clé est le journal lui-même.
- Elle occupe une petite partie de l'espace du système de fichiers.

ext2



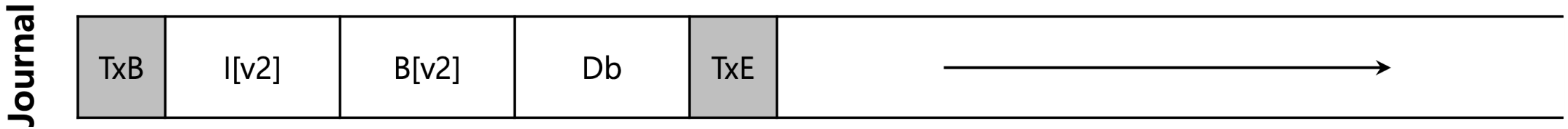
ext3



Exemple de journalisation

- Nous souhaitons mettre à jour l'inode (`I[v2]`), le bitmap (`B[v2]`) et le bloc de données (`Db`) sur le disque.
- Avant de les écrire à leur emplacement final sur le disque, nous allons d'abord les écrire dans le journal.

Entrée dans le journal



- `TxB` **(*transaction begin block*)** : le bloc de début de transaction contient un d'identifiant de transaction (TID).
- Les trois blocs du milieu contiennent simplement le contenu exact des blocs eux-mêmes → C'est ce qu'on appelle la **journalisation physique**.
- `TxE` **(*transaction end block*)** : marqueur de la fin de cette transaction, il contient également le TID.

Checkpointing

- Une fois que cette transaction est en sécurité sur le disque, nous sommes prêts à écraser les anciennes structures dans le système de fichiers.
- Ce processus est appelé **checkpointing**.
- Ainsi, pour mettre à jour le système de fichiers, nous écrivons `I[v2]`, `B[v2]` et `Db` à leurs emplacements sur le disque.

Protocole de mise à jour des données

1. **Écriture dans le journal**
    - Écriture de la transaction dans le journal et attente de la fin de ces écritures.
    - `TxB`, données en attente, mises à jour des métadonnées, `TxE`.
2. **Checkpoint**
    - Écriture des métadonnées et des mises à jour de données à leur emplacement final.

Comment organiser les écritures dans le journal ?

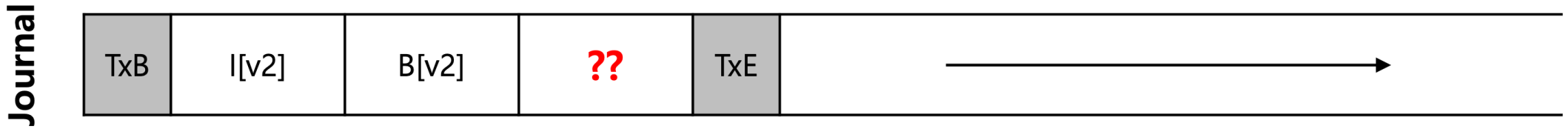
Deux approches :

1. **Un bloc à la fois**
    - 5 écritures : `TxB`, `I[v2]`, `B[v2]`, `Db`, `TxE`.
    - Lent car il faut attendre que chaque écriture soit terminée.
2. **Écrire tous les blocs en une seule fois**
    - Une seule écriture séquentielle : plus rapide.

2. Écrire tous les éléments en une seule fois (suite)

💥 Le disque peut effectuer un ordonnancement et écrire les éléments dans n'importe quel ordre.

- Le disque peut en interne (1) écrire `TxB`, `I[v2]`, `B[v2]` et `TxE` et seulement plus tard (2) écrire `Db`.
- Si le disque perd l'alimentation entre (1) et (2) :

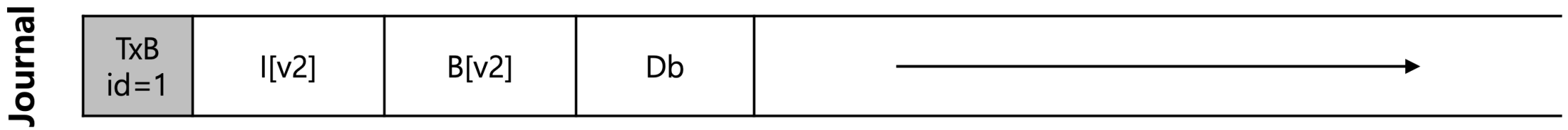


- La transaction ressemble à une transaction valide, le système de fichiers ne peut pas déterminer que le quatrième élément est erroné.
- Très problématique si cela arrive à un élément critique du système de fichiers, tel qu'un superbloc.

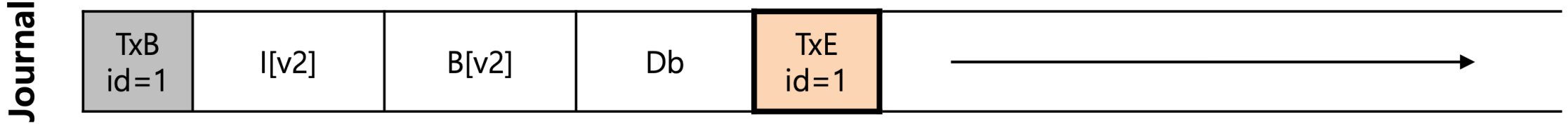
Écriture des blocs en deux étapes

Pour éviter ce problème, le système de fichiers effectue l'écriture transactionnelle en deux étapes

Tout d'abord, il écrit tous les blocs sauf le bloc `TxE` dans le journal.



Ensuite, le système de fichiers émet l'écriture du `TxE`.



Un aspect important de ce processus est la garantie d'atomicité fournie par le disque.

- Le disque garantit que toute écriture de 512 octets se produira ou ne se produira pas.
- Ainsi, `TxE` devra être un bloc unique de 512 octets.

Protocole de mise à jour des données

1. **Écriture dans le journal** : écriture du contenu de la transaction dans le journal.
2. **Commit de la transaction** : écriture du bloc de validation de la transaction.
3. **Checkpoint** : écriture des métadonnées et des mises à jour de données à leur emplacement final.

Récupération

- Si la défaillance se produit **avant que les transactions ne soient écrites** dans le journal → La mise à jour en attente est ignorée.
- Si la défaillance survient après l'**écriture des transactions dans le journal**, mais avant le checkpoint.

  On récupère la mise à jour comme suit :

  - Analyser du journal pour identifier les transactions qui ont été commitées sur le disque.
  - Les transactions sont appliquées à nouveau.

Gérer un journal de taille finie

Deux problèmes se posent lorsque le journal est plein.

1. Plus le journal est volumineux, plus la récupération sera longue.
2. Aucune autre transaction ne peut être écrite. Le système de fichiers devient inutile.

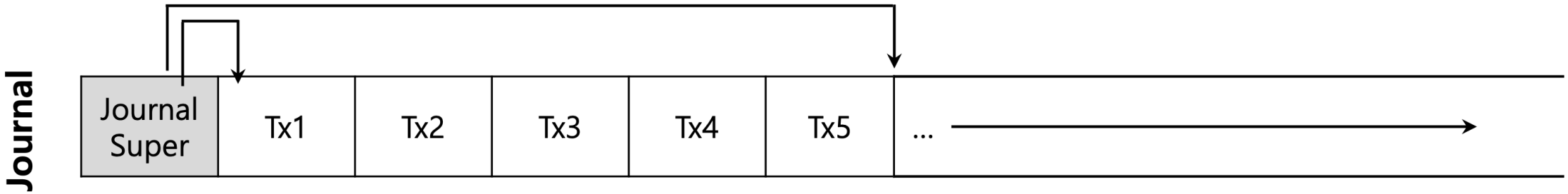
Journal circulaire

Les systèmes de fichiers traitent le journal comme une **structure de données circulaire**, qu'ils réutilisent à l'infini. On parle de journal circulaire (*circular log*).

Pour ce faire, le système de fichiers doit agir quelque temps **après un checkpoint**. Plus précisément, une fois qu'une transaction a été appliquée (*checkpointed*), le système de fichiers doit **libérer l'espace**.

Super-bloc du journal

- Permet de suivre quels sont les transactions qui été appliquées (*checkpointed*).
- Pour cela, on peut inscrire la transactions la plus ancienne et la plus récente non appliquées dans le journal ; tous le reste de l'espace est libre.



Protocole de mise à jour des données

1. Écriture dans le journal
2. Commit de la transaction
3. Checkpoint
4. **Libération** : marquer la transaction comme libre dans le journal en mettant à jour le super-bloc du journal.

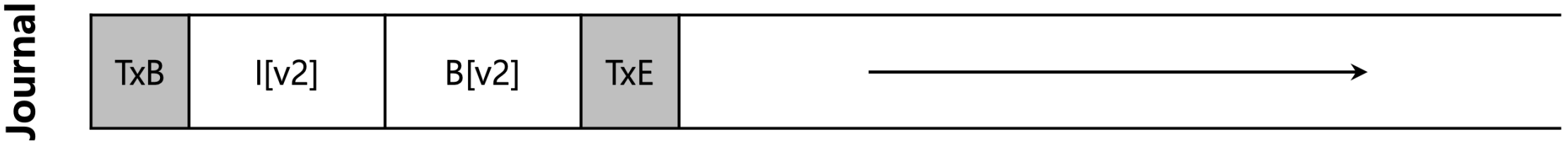
Un problème persiste : on écrit **deux fois** chaque bloc de données sur le disque.

1. Écriture du bloc de données dans le journal.
2. Checkpoint de la transaction vers un emplacement du disque.

Journalisation des métadonnées

Une forme plus simple de journalisation est appelée **journalisation des métadonnées** (ou journalisation ordonnée).

→ **Les données de l'utilisateur ne sont pas écrites dans le journal**.



Le bloc de données `Db`, précédemment écrit dans le journal, est désormais écrit à son emplacement définitif dans le système de fichiers.

Quand devons-nous écrire les blocs de données sur le disque ?

Quand devons-nous écrire les blocs de données sur le disque ?

1. **Écriture des données sur le disque après la transaction**
   💥 Cette approche pose un problème: le système de fichiers est cohérent, mais `I[v2]` peut pointer vers des données incorrectes.

2. **Écriture des données sur le disque avant la transaction**
   ✅ Cela permet d'éviter ce problème.

Protocole de mise à jour des données

1. **Écriture des données utilisateur à l'emplacement final**
2. **Écriture des métadonnées dans le journal**
3. Commit de la transaction
4. Checkpoint
5. Libération