Huitième partie

Transactions

Contenu de cette partie

- Nouvelle approche : programmation concurrente déclarative
- Mise en œuvre de cette approche déclarative : notion de transaction (issue du domaine des SGBD)
- Protocoles réalisant les propriétés de base d'un service transactionnel
  - Atomicité (tout ou rien)
  - Isolation (non interférence entre traitements)
- Adaptation de la notion de transaction au modèle de la programmation concurrente avec mémoire partagée (mémoire transactionnelle)

Plan

Interférences et isolation

Objets partagés + actions concurrentes ⇒ résultats cohérents ?
Approches :

- directe : *synchronisation* des actions contrôlant explicitement l'attente/la progression des processus (p.e. exclusion mutuelle)
- indirecte : contrôle de concurrence
  assurer un contrôle transparent assurant l'équivalence à un résultat cohérent

Contraintes d'intégrité

États cohérents décrits en intention par des contraintes d'intégrité (prédicats portant sur les valeurs des données).

Exemple

Base de données bancaire

- données = ensemble des comptes
- contraintes d'intégrité :
  - la somme des comptes est constante
  - chaque compte est positif

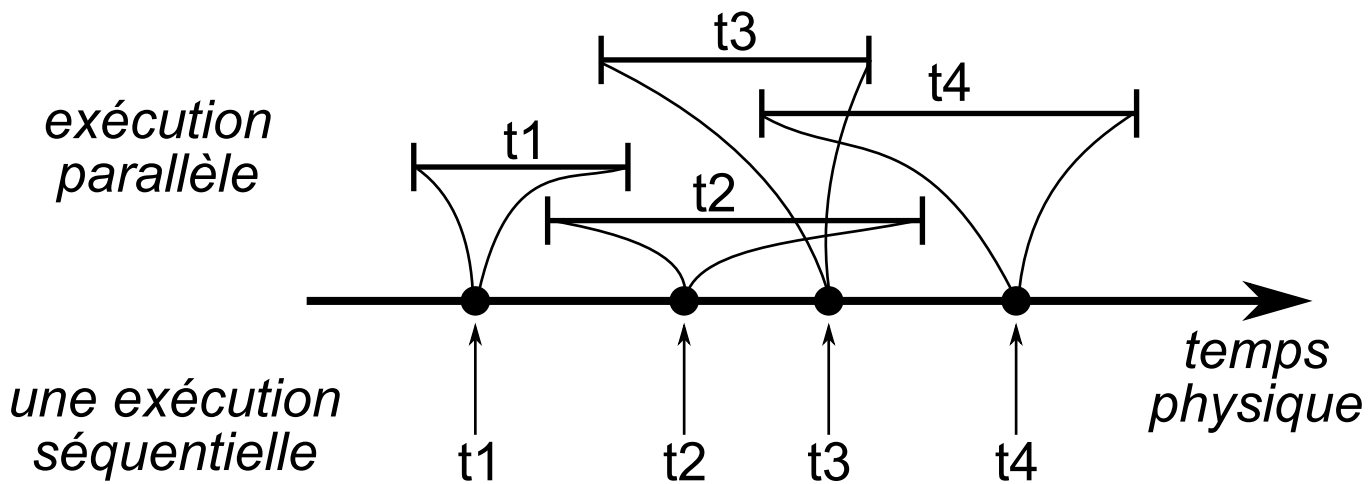
Note : les contraintes sont souvent non explicitement exprimées, l'équivalence du code concurrent avec un code séquentiel suffit.

Transaction

Définition

Suite d'opérations menant à un état cohérent, à partir de tout état cohérent.

- masquer les états intermédiaires
- parenthésage des états non observables
- possibilité d'abandon sans effet visible
  $\Rightarrow$ transaction validée (committed).



Transaction - exemple

Invariant $x + y = z$

$T1$		$T2$
1. $a_1 \leftarrow x$	$\parallel$	α. $c_2 \leftarrow z$
2. $b_1 \leftarrow y$	$\parallel$	β. $z \leftarrow c_2 + 200$
3. $x \leftarrow a_1 - 100$	$\parallel$	γ. $d_2 \leftarrow x$
4. $y \leftarrow b_1 + 100$	$\parallel$	δ. $x \leftarrow d_2 + 200$

OK : $\langle 1 \cdot 2 \cdot \alpha \cdot \beta \cdot 3 \cdot \gamma \cdot 4 \cdot \delta \rangle$ car $\equiv T1; T2$ (sérialisable)
KO : $\langle 1 \cdot 2 \cdot \alpha \cdot \beta \cdot \gamma \cdot 3 \cdot 4 \cdot \delta \rangle$ car $\not\equiv T1; T2$ et $\not\equiv T2; T1$

(x, y, z : données partagées ; a_1, b_1, c_2, d_2 variables locales privées à la transaction)

Transaction - exemple

Invariant $x = y$

$T1$		$T2$
1. $x \leftarrow x + 100$	$\parallel$	α. $x \leftarrow x * 2$
2. $y \leftarrow y + 100$	$\parallel$	β. $y \leftarrow y * 2$

OK : $\langle 1 \cdot 2 \cdot \alpha \cdot \beta \rangle$, $\langle 1 \cdot \alpha \cdot 2 \cdot \beta \rangle$, ...
KO : $\langle 1 \cdot \alpha \cdot \beta \cdot 2 \rangle$, $\langle \alpha \cdot 1 \cdot 2 \cdot \beta \rangle$

$T1$		$T2$
1. $x \leftarrow x + 100$	$\parallel$	α. $x \leftarrow x * 2$
2. $y \leftarrow y + 100$	$\parallel$	β. $y \leftarrow x$

OK : $\langle 1 \cdot 2 \cdot \alpha \cdot \beta \rangle$, $\langle 1 \cdot \alpha \cdot 2 \cdot \beta \rangle$, $\langle \alpha \cdot 1 \cdot 2 \cdot \beta \rangle$, ...
KO : $\langle 1 \cdot \alpha \cdot \beta \cdot 2 \rangle$

Propriétés ACID

Propriétés ACID

Atomicité ou « tout ou rien » : en cas d'abandon (volontaire ou subi), aucun effet visible

Cohérence : respect des contraintes d'intégrité

Isolation : pas d'interférences entre transactions = pas d'états intermédiaires observables

Durabilité : permanence des effets d'une transaction validée

Annulation/abandon

- Pour garantir la cohérence et/ou l'isolation ⇒ possibilité d'abandon (abort) d'un traitement en cours, décidé par le système de gestion.
- Du coup, autant le fournir aussi au programmeur.

Service transactionnel

Interface du service :

- `tdébut()/tfin()` : parenthésage des opérations transactionnelles
- `tabandon()` : annulation des effets de la transaction
- `técrire(...)`, `tlire(...)` : accès aux données. (opérations éventuellement implicites, mais dont l'observation est nécessaire au service transactionnel pour garantir la cohérence)

Comment évaluer la cohérence *efficacement* ?

Objectif

Eviter d'évaluer la cohérence *globalement, et à chaque instant*

- *Evaluation épisodique*/périodique (après un ensemble de pas)
  $\rightarrow$ pouvoir annuler un ensemble de pas en cas d'incohérence
- *Evaluation approchée* : trouver une condition suffisante, plus simple à évaluer (locale dans l'espace ou dans le temps)
  $\rightarrow$ notions de sérialisabilité et de conflit (cf infra)
- *Relâcher* les exigences de cohérence, afin d'avoir des critères locaux, plus simples à évaluer

Mise en œuvre

Propagation des valeurs écrites

- Contrôle la visibilité des écritures
- Optimiste (dès l'écriture) / pessimiste (à la validation)

→ Atomicité d'un ensemble d'écritures (tout ou rien)

Contrôle de concurrence

- Contrôle l'ordonnancement des opérations
- Optimiste (à la validation) / pessimiste (à chaque opération)
- Nombreuses variantes

→ Cohérence et isolation, comme si chaque transaction était seule

Ces deux politiques se combinent ± bien.

Plan

Atomicité (tout ou rien)

Objectif

- Intégrer les résultats des transactions bien terminées (= validées)
- Assurer qu'une transaction annulée n'a aucun effet sur les données partagées

Difficulté

Tenir compte de la possibilité de pannes en cours

- d'exécution,
- ou d'enregistrement des résultats définitifs,
- ou d'annulation.

Abandon sans effet

Comment abandonner une transaction sans effet ?

1. Pessimiste / propagation différée : mémoire temporaire transférée en mémoire définitive à la validation (*redo-log*)
2. Optimiste / propagation immédiate (en continu) : écriture directe avec sauvegarde de l'ancienne valeur ou de l'action inverse (journaux / *undo-log*)
   Effet domino : T' observe une écriture de T puis T abandonne $\Rightarrow$ T' doit être abandonnée

Mise en œuvre de l'atomicité

Opérations de base

- *défaire* : revenir à l'état initial d'une transaction annulée
- *refaire* : reprendre une validation interrompue par une panne

Réalisation de *défaire* et *refaire*

Basée sur la gestion d'un *journal*, conservé en *mémoire stable*.

- Contenu d'un enregistrement du journal :
  [date, id. transaction, id. objet, valeur avant (et/ou valeur après)]
- Utilisation des journaux
  - *défaire* $\rightarrow$ utiliser les valeurs avant pour revenir à l'état initial
  - *refaire* $\rightarrow$ utiliser les valeurs après pour rétablir l'état atteint
- Remarque : en cas de panne durant une opération *défaire* ou *refaire*, celle-ci peut être reprise du début.

Approche pessimiste : propagation différée

Utilisation d'un journal des valeurs après

Principe

- Écriture dans un espace de travail privé, en mémoire volatile
  → adapté aux mécanismes de gestion mémoire (caches...)
- Journalisation de la validation

- écrire → préécriture dans l'espace de travail
- valider → recopier l'espace de travail en mémoire stable (*liste d'intentions*), puis copier celle-ci en mémoire permanente
  → protection contre les pannes en cours de validation
- défaire → libérer l'espace de travail
- refaire → reprendre la recopie de la liste d'intentions

Approche optimiste : propagation en continu

Utilisation d'un journal des valeurs avant

- écrire → écriture directe en mémoire permanente
- valider → effacer les images avant
- défaire → utiliser le journal avant
- refaire → sans objet (validation sans pb)

Problèmes liés aux abandons

- Rejets en cascade

(1) técrire(x,10)	‖	(2) tlire(x)
	‖	(3) técrire(y, 8)
(4) tabandon()	‖	→ abandonner aussi

- Perte de l'état initial

initialement : x=5

(1) técrire(x,10)	‖	(2) técrire(x,8) -- sauve "x valait 10"
(3) tabandon()	‖	(4) tabandon() → x=10 au lieu de x=5

Plan

Contrôle de concurrence

Objectif

Assurer une protection contre les interférences entre transactions identique à celle obtenue avec l'exclusion mutuelle, tout en autorisant une exécution concurrente (autant que possible)

1. Exécution sérialisée : isolation par exclusion mutuelle.
2. Exécution sérialisable : contrôler l'entrelacement des actions pour que *l'effet final* soit équivalent à une exécution sérialisée.

- Il peut exister plusieurs exécutions sérialisées équivalentes
- Contrôle automatique

Sémantique

- Single-lock atomicity : exécution équivalente à l'utilisation d'un unique verrou global.
- Sérialisabilité : résultat final équivalent à une exécution sérialisée des transactions qui valident.
- Sérialisabilité stricte : sérialisabilité + respect de l'ordre temps réel (si T_A termine avant que T_B ne démarre et que les deux valident, T_A doit apparaître avant T_B dans l'exécution sérialisée équivalente)
- Linéarisabilité : transaction considérée comme une opération atomique instantanée, à un point entre son début et sa validation.
  (différence avec sérialisabilité : accès non transactionnels pris en compte)
- Opacité : sérialisabilité stricte, y compris des transactions annulées (indépendance par rapport aux transactions actives).

Conflits

Idée

- On veut que l'exécution concurrente d'un ensemble de transactions donne le même résultat qu'une exécution sérialisée de ces transactions.
- Une exécution concurrente est un entrelacement des opérations des transactions.
- L'entrelacement ne peut donner un résultat autre que s'il existe des opérations non commutatives (sinon on peut réordonner les opérations pour regrouper ensemble toutes les opérations d'une transaction).

Conflit

Opérations non commutatives exécutées sur un même objet

Exemples de conflits

Exemple

Avec opérations Lire(x) et Écrire(x,v) :

- conflit LL : non
- conflit LE : T_1.lire(x) ; . . . ; T_2.écrire(x,n) ;
- conflit EL : T_1.écrire(x,n) ; . . . ; T_2.lire(x) ;
- conflit EE : T_1.écrire(x,n) ; . . . ; T_2.écrire(x,n') ;

La notion de conflit n'est pas spécifique aux opérations Lire/Écrire :

Exemple

	lire	écrire	incrémenter	décrémenter
lire	OK	–	–	–
écrire	–	–	–	–
incrémenter	–	–	OK	OK
décrémenter	–	–	OK	OK

Graphe de dépendance, sérialisabilité

Définition

Relation de dépendance $\rightarrow$: $T_1 \rightarrow T_2$ ssi une opération de T_1 précède et est en conflit avec une opération de T_2.

Définition

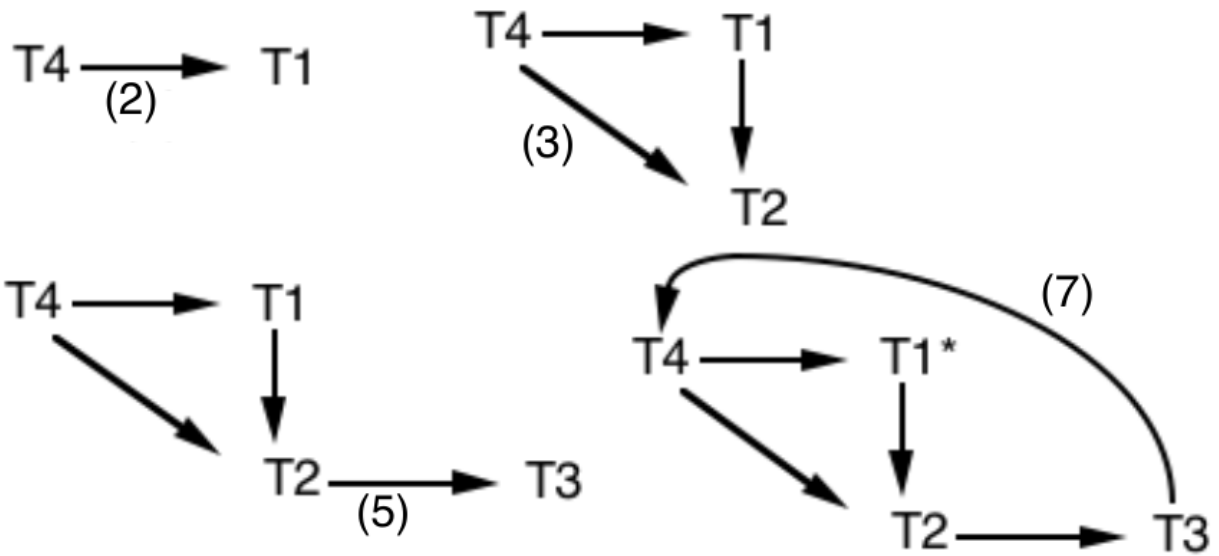
Graphe de dépendance : relations de dépendance pour les transactions déjà validées.

Théorème

Exécution sérialisable $\Leftrightarrow$ son graphe de dépendance est acyclique.

Exemple

T1	T2	T3	T4
…	…	…	(1) lire(x)
(2) écrire(x)		…	…
	(3) écrire(x)	…	…
	(4) lire(y)	(5) écrire(y)	
…	…	(6) lire(z)	(7) écrire(z)
	…	…	…
(8) valider			



sérialisation impossible ⟷ cycle

⟶ rejeter T2, ou T3, ou T4

Contrôle de concurrence

Quand vérifier la sérialisabilité :

1. à chaque terminaison d'une transaction
   (contrôle par certification ou optimiste)
2. à chaque nouvelle dépendance
   (contrôle continu ou pessimiste)

Comment garantir la sérialisabilité :

1. Utilisation explicite du graphe de dépendance
2. Fixer/observer un ordre sur les transactions qui garantit l'absence de cycle : estampilles, verrous

Certification (concurrence explicite)

Algorithme

```
-- écritures en mémoire privée avec recopie à la validation
T.lus, T.écrits : objets lus/écrits par T
T.concur : transactions ayant validé pendant l'exéc. de T
actives : ensemble des transactions en cours d'exécution
---------------------------------------------------------
procédure Certifier(T) :
si (∀ T' ∈ T.concur : T.lus ∩ T'.écrits = ∅)
alors
    -- T peut valider
    ∀ T' ∈ actives : T'.concur ← T'.concur ∪ {T}
sinon
    -- abandon de T
fin
```

Protocole coûteux et coût du rejet ⇒ faible taux de conflit

Certification (estampille)

Algorithme

```
-- écritures en mémoire privée avec recopie à la validation
C : nbre de transactions certifiées
T.déb : valeur de C au début de T
T.fin : valeur de C à la fin de T
T.val : valeur de C si T certifiée
T.lus, T.écrits : objets lus/écrits par T
```

```
procédure Certifier(T) :
si (∀ T' : T.déb < T'.val < T.fin : T.lus ∩ T'.écrits = ∅)
alors
    C ← C + 1
    T.val ← C
sinon
    abandon de T
fin
```

Protocole coûteux et coût du rejet ⇒ faible taux de conflit

Contrôle continu par estampilles

Ordre de sérialisation = ordre des estampilles

Algorithme

```
T.E : estampille de T
O.lect : estampille du plus récent lecteur de O
O.réd : estampille du plus récent écrivain de O
```

```
procédure lire(T,O)
si T.E ≥ O.réd
alors
   lecture de O possible
   O.lect ← max(O.lect,T.E)
sinon
   abandon de T
finsi
```

```
procédure écrire(T,O,v)
si T.E ≥ O.lect ∧ T.E ≥ O.réd
alors
   écriture de O possible
   O.red ← T.E
sinon
   abandon de T
finsi
```

Estampille fixée au démarrage de la transaction ou au 1er conflit.

Estampilles : abandon par prudence

Abandon superflu

T1.E = 1	T2.E = 2
lire z	
	écrire x
écrire x → abandon de T1	
	? écrire z

L'abandon n'est nécessaire que s'il y *aura* conflit effectif ultérieurement. Dans le doute, abandon.

Estampilles : amélioration

Réduire les cas d'abandon. Exemple : règle de Thomas

Algorithme

```
procédure écrire(T,O,v)
si T.E ≥ O.lect
alors
   action sérialisable : écriture possible
   si T.E ≥ O.réd
       écriture effective
       O.red ← T.E
   sinon
       rien : écriture écrasée par transaction plus récente
   finsi
sinon
   abandon de T
finsi
```

Contrôle continu par verrouillage à deux phases

Ordre de sérialisation = ordre chronologique d'accès aux objets

$T_1 \rightarrow T_2$: bloquer T_2 jusqu'à ce que T_1 valide.
Verrous en lecture/écriture/...

Si toute transaction est

- bien formée (prise du verrou avant une opération)
- à deux phases (pas de prise de verrou après une libération)

```
phase 1 : acquisitions et opérations
point de validation
phase 2 : libérations
```

alors la sérialisation est assurée.

Note : et l'interblocage ? Cf gestion de la contention.

Nécessité des deux phases

L'utilisation simple de verrous (sans règle des deux phases) ne suffit pas à assurer la sérialisation.

Invariant $x = y$

initialement $x = y = 4$

	$T1$			$T2$
1.	*lock* x	‖	α.	*lock* x
2.	$x \leftarrow x + 1$	‖	β.	*lock* y
3.	*unlock* x	‖	γ.	$x \leftarrow x * 2$
4.	*lock* y	‖	δ.	$y \leftarrow y * 2$
5.	$y \leftarrow y + 1$	‖	ϵ.	*unlock* y
6.	*unlock* y	‖	ζ.	*unlock* x

KO : $\langle 1 \cdots 3 \cdot \alpha \cdots \zeta \cdot 4 \cdots 6 \rangle$
(on obtient $x = 10, y = 9$)

Verrouillage à deux phases : justification du protocole

Idée de base

Lorsque deux transactions sont en conflit, toutes les paires d'opérations en conflit sont exécutées dans le même ordre
$\rightarrow$ pas de dépendances d'orientation opposée $\rightarrow$ pas de cycle

Schéma de preuve

- Notation :
  $e_1 \prec e_2 \equiv$ l'événement e_1 s'est produit avant l'événement e_2
- $T_i \rightarrow T_j \Rightarrow \exists O_1 : T_i.\text{libérer}(O_1) \prec T_j.\text{verrouiller}(O_1)$
- $T_j \rightarrow T_i \Rightarrow \exists O_2 : T_j.\text{libérer}(O_2) \prec T_i.\text{verrouiller}(O_2)$
- T_i à deux phases $\Rightarrow T_i.\text{verrouiller}(O_2) \prec T_i.\text{libérer}(O_1)$
- donc, T_j n'est pas à deux phases (contradiction), car :
  $T_j.\text{libérer}(O_2) \prec T_i.\text{verrouiller}(O_2) \prec T_i.\text{libérer}(O_1) \prec T_j.\textit{verrouiller}(O_1)$

Verrouillage à deux phases : interblocage

Que faire en cas de conflit de verrouillage :

- Abandon systématique ⇒ famine
- Blocage systématique ⇒ risque d'interblocage
  - ordre sur la prise des verrous (classes ordonnées)
  - prédéclaration de *tous* les verrous nécessaires (pour les prendre tous ensemble atomiquement)
- Estampilles pour prévenir les cycles dans le graphe d'attente : T_2 demande l'accès à un objet déjà alloué à T_1
  - wait-die : si T_2 a démarré avant T_1, alors bloquer T_2 sinon abandonner T_2.
    Attente permise seulement dans l'ordre des estampilles
  - wound-wait : si T_2 a démarré avant T_1 alors abandonner T_1 sinon bloquer T_2.
    Attente permise seulement dans l'ordre inverse des estampilles
    Amélioration : marquer T_1 comme *blessée* et attendre qu'elle rencontre un second conflit pour l'abandonner

Verrouillage strict à deux phases

- Prise implicite du verrou au premier accès à une variable
- Libération automatique à la validation/abandon

- Garantit simplement les deux phases
- Tout se fait à la validation : simple
- Restriction du parallélisme
  (conservation des verrous jusqu'à la fin)

Comment garantir la cohérence *efficacement* ?

Objectif

Eviter d'évaluer la cohérence *globalement, et à chaque instant*

- *Evaluation épisodique*/périodique (après un ensemble de pas)
  $\rightarrow$ pouvoir annuler un ensemble de pas en cas d'incohérence
- *Evaluation approchée* : trouver une condition suffisante, plus simple à évaluer
  $\rightarrow$ notions de sérialisabilité et de conflit
- *Relâcher* les exigences de cohérence $\rightarrow$ *cohérence faible*

Gestionnaire de contention

En cas d'abandon, détermine quand redémarrer la transaction (pour tenter d'éviter un nouvel abandon tout en garantissant la progression)

Moins que sérialisable ?

La sérialisabilité est parfois un critère trop fort → cohérence faible.
SQL définit quatre niveaux d'isolation :

- Serializable : sérialisabilité proprement dite
- Repeatable_read : lectures fantômes acceptées
- Read_committed : lectures non répétables ou fantômes acceptées
- Read_uncommitted : lectures sales, non répétables ou fantômes acceptées

Incohérences tolérables ?

Pertes de mises à jour

Écritures écrasées par d'autres écritures.

(1) a := lire(x) ;

(2) écrire(x,a+10) ;

‖

(a) b := lire(x) ;

(b) écrire(x, b+20) ;

Lectures sales

Écritures abandonnées mais observées

(1) écrire(x,100) ;

(2) abandon ;

‖

(a) b := lire(x) ;

Incohérences tolérables ?

Lectures non répétables

Donnée qui change de valeur pendant la transaction

(1) a := lire(x) ;
(2) b := lire(x) ;
$\|$
(a) écrire(x,100) ;

Lectures fantômes

Donnée agrégée qui change de contenu

(0) sum := 0 ;
(1) nb := cardinal(S)
(2) $\forall$ x $\in$ S : sum := sum + x
(3) moyenne := sum / nb
$\|$
(a) ajouter(S, 15) ;

Conclusion

- Chaque méthode a son contexte d'application privilégié
- Paramètres déterminants
  - taux de conflit
  - durée des transactions
- Résultats
  - peu de conflits → méthodes optimistes
  - nombreux conflits/transactions longues
    → verrouillage à deux phases
  - situation intermédiaire pour l'estampillage
- Simplicité de mise en œuvre du verrouillage à deux phases
  → choix le plus courant en base de données

Plan

Mémoire transactionnelle

- Introduire la notion de transactions au niveau du langage de programmation
- Objectif : se passer des verrous habituellement utilisés pour protéger les variables partagées
  - plus nécessaire d'identifier la bonne granularité des verrous
  - interblocage, etc : c'est le problème de la STM
  - gestion des priorité, etc : idem

Intégration explicite dans un langage

Exposer l'interface de manipulation des transactions et des accès.

Interface exposée

```
do {
    tx = StartTx();
    int v = tx.ReadTx(&x);
    tx.WriteTx(&y, v+1);
} while (! tx.CommitTx());
```

Intégration dans un langage

Introduire un bloc atomique :

atomically

```
atomically {
    x = y + 2;
    y = x + 3;
}
```

(analogue aux régions critiques, sans déclaration explicite des variables partagées)

Transaction et synchronisation

Synchronisation ⇒ blocage ⇒ absence de progression de la transaction ⇒ absence de progression d'autres transactions ⇒ interblocage.

Intégrer la synchronisation dans les transactions

Abandonner la transaction pour la redémarrer automatiquement quand certaines des valeurs lues auront changé.

retry

```
procédure consommer
  atomically {
      if (nbÉlémentsDisponibles > 0) {
          // choisir un élément et l'extraire
          nbÉlémentsDisponibles--
      } else {
          retry;
      }
  }
```

Transaction et synchronisation 2

Exécuter une autre transaction si la première échoue :

orelse

```
  atomically {
       // consommer dans le tampon 1
   }
orelse
  atomically {
       // consommer dans le tampon 2
  }
```

Cohérence interne

Sémantique définie sur les transactions validées (sérialisabilité) ou toutes (opacité) ?

```
                    init x=y
atomic {                          | atomic {
     if (x != y)                  |      x++;
          while (true) {}         |      y++;
}                                 | }
```

```
                int *x; bool nonnul;
atomic {                          | atomic {
     if (nonnul)                  |      x ← NULL;
          *x ← 3;                 |      nonnul ← false;
}                                 | }
```

Transaction zombie (ou condamnée) mais visible.

Interaction avec code non transactionnel

Lectures non répétables

Donnée qui change de valeur

```
atomic {                  ||
    a := lire(x);         ||
                          ||  écrire(x,100);
    b := lire(x);         ||
}                         ||
```

Lectures sales

Écritures abandonnées mais observées

```
atomic {                  ||
    écrire(x,100);        ||
                          ||  b := lire(x);
    abandon;              ||
}                         ||
```

Actions non annulables

Une transaction annulée doit être sans effet : comment faire s'il y a des effets de bords (p.e. entrées/sorties) ?

1. Interdire : uniquement des lectures/écritures de variables.
2. Ignorer le problème en considérant ces opérations comme des nop, et tant pis si la transaction est annulée.
3. Irrévocabilité : quand une transaction invoque une action non défaisable/non retardable, la transaction devient irrévocable : ne peut plus être annulée une fois l'action effectuée.
4. Intégrer dans le système transactionnel : monade d'IO d'Haskell

Transaction et exception

Exception dans une transaction ?

- Une transaction est une séquence tout ou rien de code ;
- La levée d'une exception dans un bloc doit sortir immédiatement du bloc.

1. Valider la transaction : considérer l'exception comme une branche conditionnelle.
   Simple à mettre en œuvre, ne change pas la sémantique d'un code séquentiel.
2. Annuler la transaction : considérer l'exception comme abort.
   Simplifie la composition et l'utilisation de librairie : dans `atomic { s.foo(); s.bar();}`, si `bar` échoue à cause d'une exception, rien n'a eu lieu.
   Mais si l'exception est due au code de la transaction, la cause de l'exception disparaît à l'annulation !

Imbrication

Transaction imbriquée

Transaction s'exécutant dans le contexte d'une transaction parente.

```
init x = 1
atomic{ x ← 2; atomic{ x ← x+1; abort/commit;} ...}
```

1. Une seule transaction fille active ou plusieurs (parallélisme) ?
2. Dans la fille, visibilité des effets de la transaction parente ?
3. L'annulation de la fille entraîne l'annulation de la parente ?
4. Les effets de la fille sont-ils visibles dès sa validation (ou seulement lors de la validation de la parente) ?

À plat : 3 oui, 4 non / fermée : 3 non, 4 non / ouverte : 3 non, 4 oui.

STM – Software Transactional Memory

Implantation purement logicielle de la mémoire transactionnelle.

Interface explicite

- `StartTx`, `CommitTx`, `AbortTx`
- `ReadTx(T *addr)`, `WriteTx(T *addr, T v)`,

Programmation explicite, ou insertion par le compilateur.

Points critiques

- Connaissances des accès *read-set*, *write-set*
- Journal ⇒ copie supplémentaire (*undo-log*, *redo-log*) ou double indirection (*shadow copy*)
- Meta-data associées à chaque objet élémentaire ⇒ granularité
- Efficacité

Nombreuses implantations, beaucoup de variété.

HTM – Hardware Transactional Memory

Instructions processeur

- `begin_transaction`, `end_transaction`
- Accès explicite (`load/store_transactional`) ou implicite (tous)

Accès implicite ⇒ code de bibliothèque automatiquement pris en compte + isolation forte

Implantation

- *read/write-set* : pratiquement le rôle du cache
- détection des conflits ≈ cohérence des caches
- *undo/redo-log* : dupliquer le cache

HTM – limitations

- Pas de changement de contexte pendant une transaction
- Petites transactions (2 ou 4 mots mémoire)
- Granularité fixée = unité d'accès (1 mot)
- Faux conflits dus à la granularité mot $\leftrightarrow$ ligne de cache
- Grande variété des propositions, sémantique incertaine
- Portabilité d'un code prévu pour une implantation ?

Implantation hybride

Coopération STM/HTM

- Petites transactions en HTM, grosses en STM
- Problème : détection d'inadéquation de l'HTM, basculement ?
- Problème : sémantiques différentes

Implantation STM sur HTM

- Une HTM pour petites transactions
- Implantation de la STM avec les transactions matérielles
- HTM non visible à l'extérieur

Implantation STM avec assistance matérielle

- Identifier les *bons* composants élémentaires nécessaires ⇒ implantation matérielle
- Cf Multithread / contexte CPU ou Mémoire virtuelle / MMU

Conclusion

Programmation concurrente par transactions :

- \+ simple à appréhender
- \+ réduction des bugs de programmation
- \+ nombreuses implantations portables en logiciel
- \+ compatible avec la programmation événementielle
- – nombreuses sémantiques, souvent floues (mais ce n'est pas pire que les modèles de mémoire partagée)
- – surcoût d'exécution
- – effet polluant des transactions (par transitivité sur les variables partagées)
- – questions ouvertes : code hors transaction, composition, synchronisation