Conception des systèmes concurrents

2A

3 septembre 2018

Organisation du cours

- Cours : définitions, principes, modèles
- TD : application et familiarisation avec les thèmes du cours
- TP : implémentation des schémas et principes
  - 7 TP en autonomie (Java/Ada), sur la page de l'enseignement
  - pour chaque TP :
    rendu possible (lien de dépôt, date de rendu ferme)
    → bonus
- Examen : écrit, portant sur l'ensemble de l'enseignement.

Page de l'enseignement : http ://moodle-n7.inp-toulouse.fr
Contact : mauran@enseeiht.fr

Présentation du cours

Objectif

Être capable de développer des applications parallèles (*concurrentes*)

→ modélisation pour la conception de programmes parallèles
→ connaissance des schémas (patrons) essentiels
→ raisonnement sur les programmes parallèles : exécution, propriétés
→ pratique de la programmation parallèle avec un environnement (Java) proposant les objets/outils de base

Plan du cours

1. Introduction : domaine, démarche
2. Exclusion mutuelle
3. Synchronisation à base de sémaphores
4. Interblocage
5. Synchronisation à base de moniteurs
6. Activités : Java, pthreads. . .
7. Processus communicants : CSP, Ada, C/S
8. Transactions
9. Synchronisation sans blocage

Première partie

Introduction

Contenu de cette partie

- nature et particularités des programmes concurrents
  $\Rightarrow$ conception et raisonnement systématiques et rigoureux
- modélisation des systèmes concurrents
- points clés pour faciliter la conception des applications concurrentes
- intérêt et limites de la programmation parallèle
- mise en œuvre de la programmation concurrente sur les architectures existantes

Plan

1. Le problème
   - De quoi s'agit-il ?
   - Intérêt de la programmation concurrente
   - Différences séquentiel/concurrent
2. Raisonner sur les programmes concurrents
   - Modèle d'exécution
   - Modèles d'interaction
   - Spécification des programmes concurrents
3. Conception des systèmes concurrents
   - Modularité
   - Synchronisation
4. Conclusion
5. Approfondissement : Evaluation du modèle d'entrelacement sur les architectures matérielles

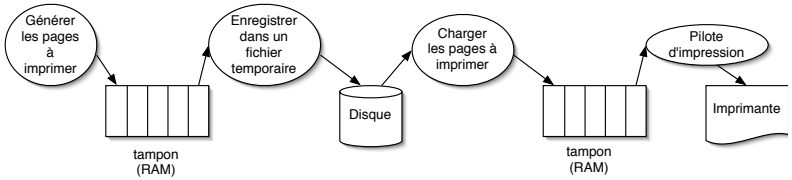
Le problème

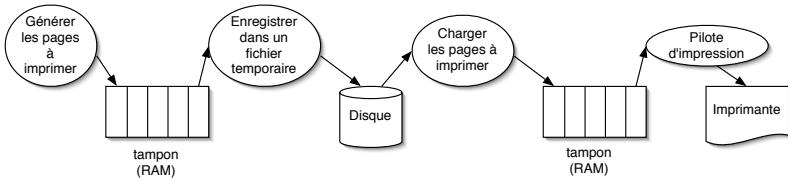
Système concurrent

Ensemble de processus s'exécutant simultanément

- en compétition pour l'utilisation de ressources partagées
- et/ou contribuant à l'obtention d'un résultat commun (global)

Exemple : service d'impression différée

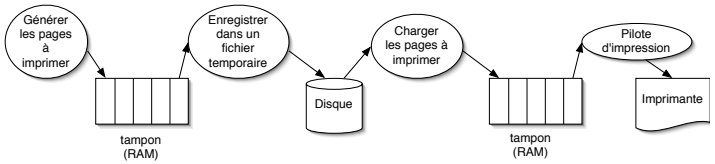




Conception : parallélisation d'un traitement

- décomposition en traitements séquentiels (*processus*)
- exécution simultanée (*concurrente*)
- les processus concurrents ne sont pas indépendants :
  ils partagent des objets (ressources, données)
  ⇒ spécifier et contrôler les interactions entre processus

Relations entre activités composées



Chaque activité progresse à son rythme, avec une vitesse arbitraire
⇒ nécessité de réaliser un couplage des activités interdépendantes

- fort : arrêt/reprise des activités ≪en avance≫ (*synchronisation*)
- faible : stockage des données échangées et non encore utilisées (*schéma producteur/consommateur*)

Expression du contrôle des interactions : 2 niveaux d'abstraction

- *coopération* (dépôt/retrait sur le tampon) :
  les activités ≪ se connaissent ≫ (interactions explicites)
- *compétition* (accès au disque) :
  les activités ≪ s'ignorent ≫ (interactions transparentes)

Intérêt de la programmation concurrente

- Facilité de conception
  le parallélisme est naturel sur beaucoup de systèmes
  - temps réel : systèmes embarqués, applications multimédia
  - mode de fonctionnement : modélisation et simulation de systèmes physiques, d'organisations, systèmes d'exploitation
- Pour accroître la puissance de calcul
  algorithmique parallèle et répartie
- Pour faire des économies
  mutualisation de ressources coûteuses via un réseau
- Parce que la technologie est mûre
  banalisation des systèmes multi-processeurs, des stations de travail/ordinateurs en réseau, services répartis

Nécessité de la programmation concurrente

- La puissance de calcul monoprocesseur atteint un plafond
  - l'augmentation des performances d'un processeur dépend directement de sa fréquence d'horloge f
    - l'énergie consommée et dissipée augmente comme f^3
      → une limite physique est atteinte depuis quelques années
  - les gains de parallélisme au niveau du processeur sont limités
    - processeurs vectoriels, architectures pipeline conviennent mal à des calculs irréguliers/généraux
  - coût excessif de l'augmentation de la taille des caches qui permettrait de compenser l'écart croissant de performances entre processeurs et mémoire
- La loi de Moore reste valide :
  la densité des transistors double tous les 18 à 24 mois

→ les architectures multiprocesseurs sont pour l'instant le principal moyen d'accroître la puissance de calcul

Qu'est-ce qui fait que la programmation concurrente est différente de la programmation séquentielle ?

- plusieurs activités simultanées ⇒ explosion de l'espace d'états

variables globales : s, i

P1	P2
s := 0	s := 0
pour i:= 1 *à* 10 *pas* 1	*pour* i:= 1 *à* 10 *pas* 1
s := s+i	s := s+i
fin_pour	*fin_pour*
afficher(s,i)	afficher(s,i)

  - P1 seul → 12 états ☺
  - P1 ‖ P2 → 12 x 12 = 144 états ☹
- interdépendance des activités
  - logique : production/utilisation de résultats intermédiaires
  - chronologique : disponibilité des résultats
  - ⇒ non déterminisme (⇒ difficulté du raisonnement par scénarios)

⇒ nécessité d'outils (conceptuels et logiciels) pour assurer le raisonnement et le développement

Plan

1. Le problème
   - De quoi s'agit-il ?
   - Intérêt de la programmation concurrente
   - Différences séquentiel/concurrent
2. Raisonner sur les programmes concurrents
   - Modèle d'exécution
   - Modèles d'interaction
   - Spécification des programmes concurrents
3. Conception des systèmes concurrents
   - Modularité
   - Synchronisation
4. Conclusion
5. Approfondissement : Evaluation du modèle d'entrelacement sur les architectures matérielles

Modèle d'exécution

Activité (ou : processus, processus léger, thread, tâche...)

- Représente l'activité d'exécution d'un programme séquentiel par un processeur
- Vision simple (simplifiée) : à chaque cycle, le processeur
  - extrait (lit et décode) une instruction machine à partir d'un flot séquentiel (le code exécutable),
  - exécute cette instruction,
  - puis écrit le résultat éventuel (registres, mémoire RAM).

→ exécution d'un processus P

= suite d'instructions effectuées $p_1; p_2; \ldots p_n$ (*histoire* de P)

Exécution concurrente

L'exécution concurrente (simultanée) d'un ensemble de processus $(P_i)_{i \in I}$ est représentée comme une exécution consistant en un *entrelacement arbitraire* des histoires de chacun des processus P_i

Exemple : 2 processus P=$p_1; p_2; p_3$ et Q=$q_1; q_2$
L'exécution concurrente de P et de Q sera vue comme (équivalente à) l'une des exécutions suivantes :
$p_1; p_2; p_3; q_1; q_2$ ou $p_1; p_2; q_1; p_3; q_2$ ou $p_1; p_2; q_1; q_2; p_3$ ou
$p_1; q_1; p_2; p_3; q_2$ ou $p_1; q_1; p_2; q_2; p_3$ ou $p_1; q_1; q_2; p_2; p_3$ ou
$q_1; p_1; p_2; p_3; q_2$ ou $q_1; p_1; p_2; q_2; p_3$ ou $q_1; p_1; q_2; p_2; p_3$ ou
$q_1; q_2; p_1; p_2; p_3$

Le modèle d'exécution par entrelacement est il réaliste ?

Abstraction réalisée

Deux instructions a et b de deux processus différents ayant une période d'exécution commune donnent un résultat identique à celui de $a; b$ ou de $b; a$

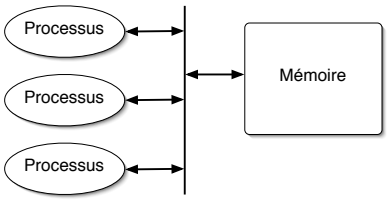
Motivation

- abstrait (ignore) les possibilités de chevauchement dans l'exécution des opérations
  ⇒ on se ramène à un ensemble *discret* de possibilités (espace d'états/produit d'histoires)
- entrelacement *arbitraire* : pas d'hypothèse sur la vitesse relative de progression des activités
  ⇒ modélise l'hétérogénéité et la charge des processeurs
- abstraction « raisonnable » au regard des architectures réelles (voir dernière section)

Modèles d'interaction : interaction par mémoire partagée

Système centralisé multi-tâches

- communication implicite, résultant de l'accès par chaque processus à des variables partagées
- processus anonymes (interaction sans identification)
- coordination (synchronisation) nécessaire (pour déterminer l'instant où une interaction est possible)



Exemples

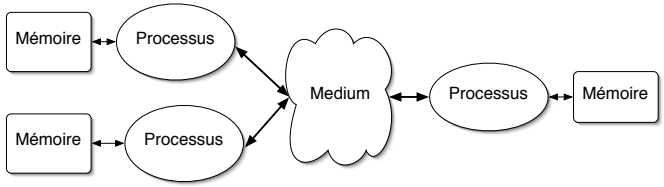
- multiprocesseurs à mémoire partagée,
- processus légers,
- Unix : couplage mémoire (mmap), fichiers

Modèles d'interaction : échange de messages

Processus communiquant par messages

Système réparti

- communication explicite par transfert de données (messages)
- désignation nécessaire du destinataire
- coordination implicite, découlant de la communication



Exemples

- processeurs en réseau,
- architectures logicielles réparties (client/serveur...),
- Unix : tubes, signaux

Spécifier un programme

Pourquoi ?

Difficulté à raisonner sur les systèmes concurrents
(explosion combinatoire de l'espace d'états/des histoires possibles)

Comment ?

Approche classique : donner les propriétés souhaitées du système, puis vérifier que ces propriétés sont valides lors des exécutions

Particularité : calculs **interdépendants** et/ou **réactifs**

→ propriétés fonctionnelles (S=f(E)) insuffisantes/inappropriées
→ propriétés sur l'évolution des traitements, au fil du temps

- Un programme est caractérisé par l'ensemble de ses exécutions possibles
- exécution = histoire, suite d'instructions/d'états (état = valeur des variables)

→ propriétés d'un programme = propriétés de ses histoires possibles

Propriété d'une histoire (suite d'états)

Validité d'un prédicat d'état

- à chaque étape de l'exécution :
  propriété de sûreté (il n'arrive jamais rien de mal)
- après un nombre de pas fini :
  propriété de vivacité (une bonne chose finit par arriver)

Exemple

- Sûreté : *Deux serveurs ne prennent jamais le même travail.*
- Vivacité : *Un travail déposé finit par être pris par un serveur*

Remarque : les propriétés exprimées peuvent porter sur

- toutes les exécutions du programme (logique temporelle linéaire)
- ou seulement certaines exécutions du programme (LT arborescente)

Les propriétés que nous aurons à considérer se limiteront généralement au cadre (plus simple) de la LT linéaire.

Vérifier les propriétés : analyse des exécutions

Définition de l'effet d'une opération : triplets de Hoare

{précondition} Opération {postcondtion}

- précondition (hypothèse) :
  propriété devant être vérifiée avant l'exécution de l'opération
- postcondition (conclusion) :
  propriété garantie par l'exécution de l'opération

Exemple

$\{t = \text{nb requêtes en attente} \wedge t > 0 \wedge r = \text{nb résultats}\}$
le serveur traite une requête
$\{\text{nb requêtes en attente} = t - 1 \wedge \text{nb résultats} = r + 1\}$

Analyse d'une exécution

- partir d'une propriété (hypothèse) caractérisant l'état initial
- appliquer en séquence les opérations de l'histoire :
  propriété établie par l'exécution d'une op. = précondition de l'op. suivante

Analyse des exécutions : propriétés d'actions concurrentes

Propriétés établies par la combinaison des actions (exemples)

Sérialisation (sémantique de l'entrelacement) :

$$\frac{\{p\}A_1; A_2\{q_{12}\},\ \{p\}A_2; A_1\{q_{21}\}}{\{p\}A_1 \parallel A_2\{q_{12} \vee q_{21}\}}$$

Indépendance (des effets de calculs séparés) :

$$\frac{A_1 \text{et } A_2 \text{ sans interférence},\ \{p\}A_1\{q_1\},\ \{p\}A_2\{q_2\}}{\{p\}A_1 \parallel A_2\{q_1 \wedge q_2\}}$$

Plan

1 Le problème
- De quoi s'agit-il ?
- Intérêt de la programmation concurrente
- Différences séquentiel/concurrent

2 Raisonner sur les programmes concurrents
- Modèle d'exécution
- Modèles d'interaction
- Spécification des programmes concurrents

3 Conception des systèmes concurrents
- Modularité
- Synchronisation

4 Conclusion

5 Approfondissement : Evaluation du modèle d'entrelacement sur les architectures matérielles

Conception des systèmes concurrents

Point clé :
contrôler les effets des interactions/interférences entre processus

- isoler (raisonner indépendamment) → modularité
- contrôler/spécifier l'interaction
  - définir les instants où l'interaction est possible
  - relier ces instants au flot d'exécution de chacun des processus

Modularité : pouvoir raisonner sur chaque activité séparément

Atomicité

mécanisme/protocole garantissant qu'une (série d')opération(s) est exécutée complètement et sans interférence (isolément)

- grain fin (instruction)
  - (modèle) utile pour le raisonnement : entrelacement
  - (matériel) utile pour déterminer un résultat en cas de conflit
- gros grain (bloc d'instructions) : utile pour la conception.

Réalisation directe :
exclusion mutuelle (bloquer tous les processus sauf 1)

- verrous
- masquage des interruptions (sur un monoprocesseur)
- ...

Contrôle des interactions : synchronisation

Mise en œuvre : attente

Un processus prêt pour une interaction est mis en attente (bloqué), jusqu'à ce que **tous** les processus participants soient prêts.

Expression

- en termes de
  - flot de contrôle : placer un *point de synchronisation commun* dans le code de chacun des processus d'un groupe de processus. Ce point de synchronisation définira un instant d'exécution commun à ces processus.
  - flot de données : définir les *échanges* de données entre processus (émission/réception de messages, ou d'événements). L'ordonnancement des processus suit la circulation de l'information.
- globale (barrière, événements, invariants) ou individuelle (rendez-vous, canaux)

Comment pouvoir raisonner sur chaque interaction séparément ? (1/3)

Principe

Définir les interactions permises, indépendamment des calculs

Première idée

Spécifier les suites d'interactions possibles (légales) pour les activités

→ grammaire définissant les suites d'opérations (interactions) permises (expressions de chemins)

→ moyen de vérifier de manière simple et indépendante du code des processus si 1 exécution (trace) globale est correcte (légale)

Exemple : interaction client/serveur

A tout moment, *nb d'appels à déposer_tâche* $\geq$ *nb d'appels à traiter_tâche*

Difficulté

Composition (ajout/retrait d'opérations ⇒ redéfinir les suites)

Comment pouvoir raisonner sur chaque interaction séparément ? (2/3)

Deuxième étape

Définir les interactions permises, indépendamment des opérations

Idée

Les processus doivent se synchroniser parce qu'il **partagent** un objet

- à construire (coopération)
- à utiliser (concurrence)

→ spécifier un objet partagé, caractérisé par un ensemble d'états possibles (légaux) : invariant portant sur l'état de l'objet partagé

Exemple : *la file des travaux à traiter peut contenir de* 0 *à Max travaux*

→ indépendance par rapport aux opérations des processus
(Les interactions correctes sont celles qui maintiennent l'invariant)

Difficulté

Nécessite de connaître l'invariant (OK pour un système fermé)

Comment pouvoir raisonner sur chaque interaction séparément ? (3/3)

Systèmes ouverts

Situation : tous les processus ne sont pas connus à l'avance (au moment de la conception)

→ définition de critères de cohérence :

- proposer 1 interface d'accès aux objets partagés, permettant de
- contrôler (automatiquement) les accès pour garantir une propriété globale sur le résultat de l'exécution, indépendamment de l'ordre d'exécution réel

Exemples

- Equivalence à une exécution en exclusion mutuelle
  → maintien de tout invariant : mémoire transactionnelle
- Equivalence à une exécution entrelacée : cohérence mémoire

Plan

1. Le problème
   - De quoi s'agit-il ?
   - Intérêt de la programmation concurrente
   - Différences séquentiel/concurrent
2. Raisonner sur les programmes concurrents
   - Modèle d'exécution
   - Modèles d'interaction
   - Spécification des programmes concurrents
3. Conception des systèmes concurrents
   - Modularité
   - Synchronisation
4. Conclusion
5. Approfondissement : Evaluation du modèle d'entrelacement sur les architectures matérielles

Bilan

\+ modèle de programmation naturel

– surcoût d'exécution (synchronisation, implantation du pseudo-parallélisme).

– surcoût de développement : nécessité d'expliciter la synchronisation, vérifier la réentrance des bibliothèques, danger des variables partagées.

– surcoût de mise-au-point : débogage souvent délicat (pas de flot séquentiel à suivre, non déterminisme) ; effet d'interférence entre des activités, interblocage...

\+ **parallélisme (répartition ou multiprocesseurs) = moyen actuel privilégié pour augmenter la puissance de calcul**

Parallélisme et performance

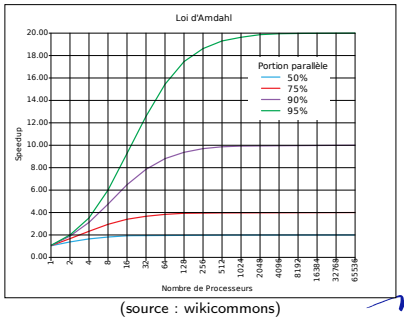
Idée naïve sur le parallélisme

≪ *Si je remplace ma machine mono-processeur par une machine à N processeurs, mon programme ira N fois plus vite* ≫

Soit un système composé par une partie p parallélisable + une partie $1-p$ séquentielle.

CPU	durée	$p = 40\%$	$p = 80\%$
1	$p + (1-p)$	100	100
4	$\frac{p}{4} + (1-p)$	70	40
8	$\frac{p}{8} + (1-p)$	65	30
16	$\frac{p}{16} + (1-p)$	62,5	25
∞	$0 + (1-p)$	60	20

Loi d'Amdahl :
facteur d'accélération maximal = $\frac{1}{1-p}$



(source : wikicommons)

Parallélisme et performance

Idée naïve sur la performance

≪ *Si je remplace ma machine par une machine N fois plus rapide, mon programme traitera des problèmes N fois plus grands dans le même temps* ≫

Pour un problème de taille n soluble en temps T, taille de problème soluble dans le même temps sur une machine N fois plus rapide :

complexité	$N = 4$	$N = 16$	$N = 1024$
$O(n)$	$4n$	$16n$	$1024n$
$O(n^2)$	$\sqrt{4}n = 2n$	$\sqrt{16}n = 4n$	$\sqrt{1024}n = 32n$
$O(n^3)$	$\sqrt[3]{4}n \approx 1.6n$	$\sqrt[3]{16}n \approx 2.5n$	$\sqrt[3]{1024}n \approx 10n$
$O(e^n)$	$\ln(4)n \approx 1.4n$	$\ln(16)n \approx 2.8n$	$\ln(1024)n \approx 6.9n$

En supposant en outre que tout est 100% est parallélisable et qu'il n'y a aucune interférence !

Plan

1 Le problème
- De quoi s'agit-il ?
- Intérêt de la programmation concurrente
- Différences séquentiel/concurrent

2 Raisonner sur les programmes concurrents
- Modèle d'exécution
- Modèles d'interaction
- Spécification des programmes concurrents

3 Conception des systèmes concurrents
- Modularité
- Synchronisation

4 Conclusion

5 Approfondissement : Evaluation du modèle d'entrelacement sur les architectures matérielles

Evaluation : architecture monoprocesseur

Modèle d'exécution abstrait : entrelacement

L'exécution concurrente (simultanée) d'un ensemble de processus $(P_i)_{i \in I}$ est représentée comme une exécution consistant en un *entrelacement arbitraire* des histoires de chacun des processus P_i

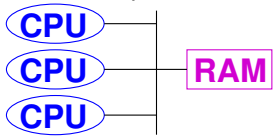
Réalisation sur un monoprocesseur

Pseudo parallélisme (ou parallélisme virtuel)

- le processeur est alloué à tour de rôle à chacun des processus par l'ordonnanceur du système d'exploitation
- le modèle reflète la réalité
- le parallélisme garde tout son intérêt comme
  - outil de conception et d'organisation des traitements,
  - et pour assurer une indépendance par rapport au matériel.

Evaluation : multiprocesseurs SMP (vrai parallélisme)

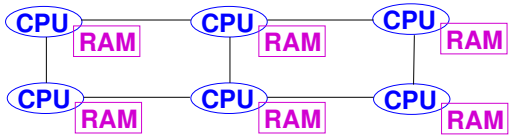
[SMP] Symmetric MultiProcessor :
une mémoire + un ensemble de processeurs



- tant que les processus travaillent sur des zones mémoires distinctes $a; b$ ou $b; a$ ou encore une exécution réellement simultanée de a et b donnent le même résultat
- si a et b opèrent simultanément sur une même zone mémoire, le résultat serait imprévisible, *mais* les requêtes d'accès à la mémoire sont (en général) traitées en séquence par le matériel, pour une taille de bloc donnée.
  Le résultat sera donc le même que celui de $a; b$ ou de $b; a$
- le modèle reflète donc la réalité

Evaluation : multiprocesseurs NUMA (vrai parallélisme)

[NUMA] : Non-Uniform Memory Access
graphe d'interconnexion de {CPU+mémoire}



- chaque nœud/site opère sur sa mémoire locale, et traite en séquence les requêtes d'accès à sa mémoire locale provenant d'autres sites/nœuds
- le modèle reflète donc la réalité

Modèle et réalité : un bémol

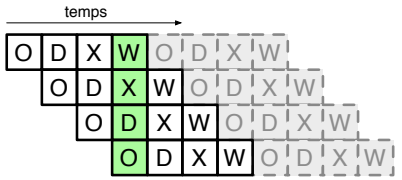
Les architectures récentes éloignent le modèle de la réalité :

- au niveau du processeur : fragmentation et concurrence à grain fin
  - pipeline : plusieurs instructions en cours dans un même cycle : obtention, décodage, exécution, écriture du résultat
  - superscalaire : plusieurs unités d'exécution (et pipeline)
  - instructions vectorielles
  - réordonnancement (out-of-order)
- au niveau de la mémoire : utilisation de caches

Concurrence à grain fin : pipeline

Principe

- chaque instruction comporte une série d'étapes :
  obtention (O)/décodage (D)/exécution (X)/écriture du résultat (W)
- chaque étape est traitée par un circuit à part
- le pipeline permet de charger plusieurs instructions et ainsi d'utiliser simultanément les circuits dédiés, chacun opérant sur une instruction



Difficulté

dépendances entre données utilisées par des instructions proches

```
ADD R1, R1,1    # R1++
SUB R2, R1, 10  # R2 := R1 - 10
```

Remèdes

- insertion de NOP (bulles) pour limiter le traitement parallèle
- réordonnancement (éloignement) des instructions dépendantes

Caches

La mémoire et le processeur sont éloignés : un accès mémoire est considérablement plus lent que l'exécution d'une instruction (peut atteindre un facteur 100 dans un ordinateur, 10000 en réparti).
Principe de localité :

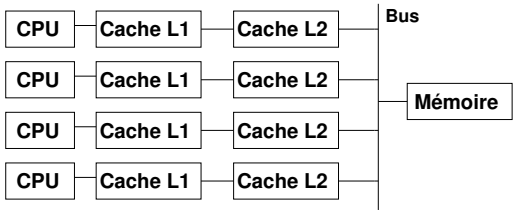
- temporelle : si on utilise une adresse, on l'utilisera probablement de nouveau dans peu de temps
- spatiale : si on utilise une adresse, on utilisera probablement une adresse proche dans peu de temps

⇒ conserver près du CPU les dernières cases mémoire accédées
⇒ Cache : mémoire rapide proche du processeur

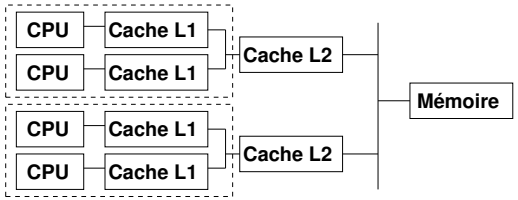
Plusieurs niveaux de caches : de plus en plus gros, de moins en moins rapides (couramment 3 niveaux).

Caches sur les architectures à multi-processeurs

Multi-processeurs ≪ à l'ancienne ≫ :



Multi-processeurs multi-cœurs :



Problème :
cohérence/arbitrage si plusieurs copies en cache d'un même mot mémoire

Comment fonctionne l'écriture d'une case mémoire avec les caches ?

Write-Through diffusion sur le bus à chaque valeur écrite
- \+ visible par les autres processeurs ⇒ invalidation des valeurs passées
- \+ la mémoire et le cache sont cohérents
- − trafic inutile : écritures répétées, écritures de variables privées au thread

Write-Back diffusion uniquement à l'éviction de la ligne
- \+ trafic minimal
- − cohérence cache - mémoire - autres caches

Cohérence mémoire

Si un processeur écrit la case d'adresse a_1, quand les autres processeurs verront-ils cette valeur ? Si plusieurs écritures consécutives en a_1, $a_2 \ldots$, sont-elles vues dans cet ordre ?

Règles de cohérence mémoire

Cohérence séquentielle le résultat d'une exécution parallèle est le même que celui d'une exécution séquentielle qui respecte l'ordre partiel de chacun des processeurs.

Cohérence PRAM (pipelined RAM ou fifo) les écritures d'un même processeur sont vues dans l'ordre où elles ont été effectuées ; des écritures de processeurs différents peuvent être vues dans des ordres différents.

Cohérence ≪ lente ≫ (slow consistency) : une lecture retourne *une* valeur précédemment écrite, sans remonter dans le temps.

Cohérence Mémoire – exemple

Init : `x = 0` $\wedge$ `y = 0`

Processeur P1	Processeur P2
(1) `x` $\leftarrow$ `1`	(a) `y` $\leftarrow$ `1`
(2) `t1` $\leftarrow$ `y`	(b) `t2` $\leftarrow$ `x`

Un résultat `t1 = 0` $\wedge$ `t2 = 0` est possible en cohérence PRAM et slow, impossible en cohérence séquentielle.

Le mot de la fin

Les mécanismes disponibles sur les architectures actuelles permettent d'accélérer l'exécution de traitements indépendants, mais n'offrent pas de garanties sur la cohérence du résultat de l'exécution d'activités coordonnées/interdépendantes

- contrôler/débrayer ces mécanismes
  - vidage des caches
  - inhibition des caches ($\approx$ variables `volatile` en Java)
  - remplissage des pipeline
  - choix de protocoles de cohérence mémoire
- préciser les hypothèses faites sur le matériel par les différents protocoles de synchronisation
  Exemple : accès séquentiels sur les variables partagées

Deuxième partie

Protocoles d'exclusion mutuelle

Contenu de cette partie

- difficultés résultant d'accès concurrents à un objet partagé
- mise en œuvre de protocoles d'isolation
  - solutions synchrones (i. e. bloquantes) : attente active
    - → difficulté du raisonnement en algorithmique concurrente
    - → aides fournies au niveau matériel
  - solutions asynchrones : gestion des processus

Plan

1 Interférences entre actions
- Isolation
- Protocoles d'exclusion mutuelle

2 Mise en œuvre
- Solutions logicielles
- Solutions matérielles
- Peut-on se passer d'attente active ?
- En pratique. . .

Interférences et isolation

S_1

```
(1) x := lire(compte_2);
(2) y := lire(compte_1);
(3) y := y + x;
(4) ecrire(compte_1, y);
```

S_2

```
(a) v := lire(compte_1);
(b) v := v - 100;
(c) ecrire(compte_1, v);
```

- compte_1 et compte_2 sont partagés par les deux traitements ;
- les variables x, y et v sont locales à chacun des traitements ;
- les traitements s'exécutent en parallèle, et leurs actions peuvent être entrelacées.

(1) (2) (3) (4) (a) (b) (c) est une exécution possible, cohérente.
(1) (a) (b) (c) (2) (3) (4) " " " " "
(1) (2) (a) (3) (b) (4) (c) est une exécution possible, incohérente .

cohérence ⇐ calculs séparés ⇐ exécution séquentielle

Section critique

Définition

Les séquences $S_1 = \texttt{(1);(2);(3);(4)}$ et $S_2 = \texttt{(a);(b);(c)}$ sont des sections critiques, qui sont chacune destinées à être exécutées de manière atomique (indivisible) :

- le résultat de l'exécution concurrente de S_1 et S_2 doit être le même que celui de l'une des exécutions séquentielles $S_1; S_2$ ou bien $S_2; S_1$.
- cette équivalence peut être atteinte
  - en contrôlant directement l'ordre d'exécution de S_1 et S_2 (exclusion mutuelle),
  - ou en contrôlant les résultats (partiels ou finaux) de S_1 et S_2 (contrôle de concurrence).

Accès conflictuels : encore des exemples

Exécution concurrente

```
init x = 0;
⟨ a := x; x := a + 1 ⟩ ‖ ⟨ b := x; x := b - 1 ⟩
⇒ x = -1, 0 ou 1
```

Modifications concurrentes

⟨ x := 0x|00|01| ⟩ ‖ ⟨ x := 0x|02|00| ⟩
⇒ x = 0x0001 ou 0x0200 ou 0x0201 ou 0x0000 ou 1234 !

Cohérence mémoire

```
init x = 0 ∧ y = 0
⟨ x := 1; y := 2 ⟩ ‖ ⟨ printf("%d %d",x,y); ⟩
⇒ affiche 0 0 ou 1 2 ou 1 0 ou 0 2 ...
```


Protocoles d'exclusion mutuelle : contexte

- ensemble de processus concurrents P_i
- variables partagées par tous les processus
  variables privées (locales) à chaque processus
- structure de chacun des processus

```
cycle
      entrée   section critique   sortie
      ⋮
fincycle
```

- hypothèses :
  - vitesse d'exécution non nulle
  - section critique de durée finie

Objectif

Garantir l'exécution en exclusion mutuelle des $\neq$ sections critiques

Protocoles d'exclusion mutuelle : propriétés

- (sûreté) à tout moment, au plus un processus est en cours d'exécution d'une section critique (noté $P_k.excl$)

$$\textbf{invariant } \forall i, j \in 0..N-1 : P_i.excl \wedge P_j.excl \Rightarrow i = j$$

- (vivacité faible) lorsqu'il y a (au moins) une demande ($P_k.dem$), un processus qui demande à entrer sera admis

$$\forall i \in 0..N-1 : (P_i.dem \textbf{ leadsto } \exists j \in 0..N-1 : P_j.excl)$$

- (vivacité forte) si un processus demande à entrer, ce processus finira par obtenir l'accès (son attente est finie)

$$\forall i \in 0..N-1 : P_i.dem \textbf{ leadsto } P_i.excl$$

Plan

1 Interférences entre actions
- Isolation
- Protocoles d'exclusion mutuelle

2 Mise en œuvre
- Solutions logicielles
- Solutions matérielles
- Peut-on se passer d'attente active ?
- En pratique...

Mise en œuvre : moyens

- Solutions directes
  (plutôt synchronisation à grain fin)
  - solutions logicielles : lecture/écriture de variables partagées
    → attente active : tester continûment la possibilité de progresser
  - mécanismes matériels
    - simplifiant l'attente active (instructions spécialisées)
    - évitant l'attente active (masquage des interruptions)
- Recours au service de gestion des activités de l'environnement d'exécution (système d'exploitation...)

Solutions logicielles : premier essai

Algorithme

```
occupé : global boolean := false;
tant que occupé faire nop;
occupé ← true;
    section critique
occupé ← false;
```

Problème
Lecture (test) et écriture (affectation) effectuées séparément
→ invariant invalide

Solutions logicielles : alternance

Deux processus (P_0 et P_1)

Algorithme (code du processus P_i)

```
tour : global 0..1;
tant que tour ≠ i faire nop;
    section critique
tour ← i + 1 mod 2;
```

- lectures et écritures supposées atomiques
- généralisable à plus de 2 processus

Problème
alternance obligatoire

Solutions logicielles : priorité à l'autre demandeur

Deux processus (P_0 et P_1)

Algorithme (code de P_i, avec j = id. de l'autre processus)

```
demande : global array 0..1 of boolean;
demande[i] ← true;
tant que demande[j] faire nop;
    section critique
demande[i] ← false;
```

- lectures et écritures supposées atomiques
- non facilement généralisable à plus de 2 processus

Problème
risque d'attente infinie (interblocage)

Solutions logicielles : Peterson 1981 (1/2)

Deux processus (P_0 et P_1)

Algorithme (code de P_i, avec j = id. de l'autre processus)

```
demande : global array 0..1 of boolean;
tour : global 0..1;
demande[i] ← true;
tour ← j;
tant que (demande[j] et tour = j) faire nop;
    section critique
demande[i] ← false;
```

- lectures et écritures supposées atomiques
- évaluation non atomique du ≪ et ≫
- vivacité forte

Solutions logicielles : Peterson 1981 (2/2)

Exercice

L'ordre des deux premières instructions du protocole d'entrée est-il important ? Pourquoi ?

Idée de la preuve

- sûreté
  - *tour* ne peut avoir qu'une valeur (et *tour* n'est pas modifié dans la section critique)
- vivacité forte
  - si P_i attend, (*demande[j]* et *tour* = j) finit par devenir et rester faux

Solution logicielle pour n processus (Lamport 1974)

L'algorithme de la boulangerie (code du processus P_i)

```
choix : global array 0..N-1 of boolean;
num : global array 0..N-1 of integer;
tour : integer;
choix[i] ← true;
tour ← 0;
pour k de 0 à N faire tour ← max(tour,num[k]);
num[i] ← tour + 1;
choix[i] ← false;

pour k de 0 à N faire
 tant que (choix[k]) faire nop;
 tant que (num[k]≠0) et (num[k],k)≺(num[i],i) faire nop;
section critique
num[i] ← 0;
```

Remarque : autorise des lectures et écriture non atomiques

Solutions matérielles : instructions spécifiques

TestAndSet(x), instruction

- indivisible
- qui positionne x à vrai
- et renvoie l'ancienne valeur de x

Définition

```
function TestAndSet (x : in out boolean) : boolean
     declare oldx : boolean
begin
     oldx := x; x := true; //instruction atomique
     return oldx;
end TestAndSet
```

Solutions matérielles : utilisation du TestAndSet

Algorithme

```
occupé : global boolean := false;
tant que TestAndSet(occupé) faire nop;
  section critique
occupé ← false;
```

Tous les processeurs actuels possèdent une instruction analogue au TestAndSet, et adaptée aux multi-processeurs symétriques.

Question
Ce protocole est-il vivace ?

Solutions matérielles : utilisation de FetchAndAdd

Définition

```
function FetchAndAdd (x : in out int) : int
     declare oldx : int
begin
     oldx := x; x := oldx + 1;
     return oldx;
end FetchAndAdd
```

```
ticket : global int := 0;
tour : global int := 0;
montour : local int;
montour ← FetchAndAdd(ticket);
tant que tour ≠ montour faire nop;
     section critique
FetchAndAdd(tour);
```

Question : ce protocole est-il vivace ?

Solution matérielle sans matériel : utilisation du système de fichiers

Les primitives du noyau Unix permettant la création conditionnelle de fichiers peuvent être utilisées comme des opérations atomiques analogues au TestAndSet.

Algorithme

```
tant que
     open("toto", O_RDONLY | O_EXCL | O_CREAT, 0) == -1
faire nop;
     section critique
unlink("toto");
```

- ne nécessite pas de mémoire partagée
- atomicité assurée par le noyau d'exécution

Peut-on se passer d'attente active ?

Les solutions précédentes sont correctes,
mais présentent un inconvénient sérieux

Attente active

Un processus demandant la section critique et la trouvant occupée doit tester en permanence la possibilité d'entrer en section critique

→ monopolisation ≪ inutile ≫ du (temps) processeur

Piste d'amélioration

Éviter qu'un processus devant attendre en entrée de la section critique répète ces tests ≪ inutiles ≫...

Éviter l'attente active : recours à la gestion des activités

Algorithme

(<< B >> indique que le bloc d'instructions B doit être exécuté en exclusion mutuelle)

```
occupé : global bool := false;
demandeurs : global fifo;
```

```
<< si occupé alors
      self ← identifiant du processus courant
      ajouter self dans demandeurs
      se suspendre
   sinon
      occupé ← true
   finsi >>
    section critique
<< si demandeurs est non vide alors
      p ← extraire premier de demandeurs
      débloquer p
   sinon
      occupé ← false
   finsi >>
```

- accès aux variables globales (*demandeurs*, *occupé*) en exclusion mutuelle
- cette exclusion mutuelle est réalisée par attente active (acceptable, car sections critiques courtes)

Solution matérielle : masquage des interruptions

Idée : réserver le processeur au processus en section critique

Algorithme

```
masquer les interruptions
  section critique
démasquer les interruptions
```

Limite importante :

ne fonctionne que sur les mono-processeurs
→ pas d'entrée-sortie, pas de défaut de page en SC
→ micro-systèmes embarqués

Éviter l'attente active : utilisation des primitives de verrouillage de fichiers

Verrous

- exclusifs (appel système lockf)
- ou coopératifs sur les fichiers : en lecture partagée, en écriture exclusive (appels système : flock, fcntl)

Algorithme

```
fd = open ("toto", O_RDWR);
lockf (fd, F_LOCK, 0); // verrouillage exclusif
    section critique
lockf (fd, F_ULOCK, 0); // déverrouillage
```

- attente passive (le processus est bloqué)
- portabilité aléatoire

En pratique...

- L'attente active ne peut être éliminée, mais est à limiter le plus possible sur un monoprocesseur ; elle peut être utile pour la synchronisation à grain fin sur les multiprocesseurs
- La plupart des environnements d'exécution offrent un service analogue aux verrous, avec les opérations atomiques :
  - obtenir (acquire) : si le verrou est libre, l'attribuer au processus demandeur ; sinon bloquer le processus demandeur
  - libérer (release) : si au moins un processus est en attente du verrou, transférer la possession à l'un des demandeurs et le débloquer ; sinon marquer le verrou comme libre.

Algorithme

```
accès : global verrou // partagé
obtenir accès
     section critique
libérer accès
```


Exercice : exclusion mutuelle vivace avec TestAndSet

Algorithme

```
occupé : global booléen := faux;
attend : global tableau 0..N-1 de booléen := faux;

//Code du processus i
j : local 0..N-1;
attend[i] := vrai;
tant que (attend[i] et TestAndSet(occupé)) faire nop;
     section critique
attend[i] := faux;
j := i+1 % N;
tant que (i ≠ j et non attend[j]) faire j := j+1 % N;
si j = i alors occupé := faux;
          sinon attend[j] := faux ;
```

Troisième partie

Sémaphores

Contenu de cette partie

- présentation d'un objet de synchronisation « minimal » (sémaphore)
- patrons de conception élémentaires utilisant les sémaphores
- exemple récapitulatif (schéma producteurs/consommateurs)
- schémas d'utilisation pour le contrôle fin de l'accès aux ressources partagées
- mise en œuvre des sémaphores

Plan

1. Spécification
   - Introduction
   - Définition
   - Modèle intuitif
   - Remarques
2. Utilisation des sémaphores
   - Schémas de base
   - Schéma producteurs/consommateurs
   - Contrôle fin de l'accès concurrent aux ressources partagées
3. Mise en œuvre des sémaphores
   - Utilisation des la gestion des processus
   - Sémaphore général à partir de sémaphores binaires
   - L'inversion de priorité

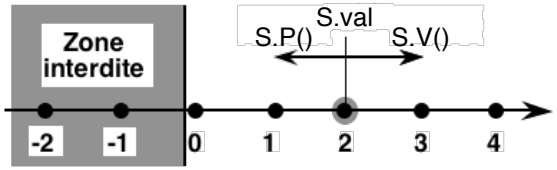
But

- Fournir un moyen *simple*, élémentaire, de contrôler les effets des interactions entre processus
  - isoler (modularité) : atomicité
  - spécifier des interactions précises : synchronisation
- Exprimer ce contrôle par des interactions sur un *objet partagé* (indépendant des processus en concurrence) plutôt que par des interactions entre processus (dont le code et le comportement seraient alors interdépendants)

Définition – Dijkstra 1968

Un sémaphore S est un objet dont

- l'état val est un attribut entier *privé* (l'état est encapsulé)
- l'ensemble des états permis est contraint par un invariant (*contrainte de synchronisation*) :
  invariant $S.val \geq 0$ (l'état doit toujours rester positif ou nul)
- l'interface fournit deux opérations principales :
  - P : bloque si l'état est nul, décrémente l'état lorsqu'il est > 0
  - V : incrémente l'état
    $\rightarrow$ permet le déblocage d'un éventuel processus bloqué sur P
  - les opérations `P` et `V` sont *atomiques*



Compléments

- Autres noms des opérations :
  - P : down, wait/attendre, acquire/prendre
  - V : up, signal/signaler, release/libérer
- Autre opération : création (et/ou initialisation)
  $S = newSemaphore(v_0)$ (ou $S.init(v_0)$)
  (crée et) initialise l'état de S à v_0
- Si la précondition de $S.P()$ (c'est-à-dire $S.val > 0$) n'est pas vérifiée, le processus est retardé ou bloqué.
- l'invariant du sémaphore peut aussi s'exprimer à partir des nombres $\#P$ et $\#V$ d'opérations P et V effectuées :
  invariant $S.val = S.val_{\text{init}} + \#V - \#P$

Modèle intuitif

Un sémaphore peut être vu comme un tas de jetons avec 2 actions

- Prendre un jeton, en attendant si nécessaire qu'il y en ait ;
- Déposer un jeton.

Attention

- les jetons sont anonymes et illimités : un processus peut déposer un jeton sans en avoir pris ;
- il n'y a pas de lien entre le jeton déposé et le processus déposeur ;
- lorsqu'un processus dépose un jeton et que des processus sont en attente, *un seul* d'entre eux peut prendre ce jeton.

Remarques

1. Lors de l'exécution d'une opération V, s'il existe plusieurs processus en attente, la politique de choix du processus à débloquer peut être :
   - par ordre chronologique d'arrivée (FIFO) : équitable
   - associée à une priorité affectée aux processus en attente
   - indéfinie.
     C'est le cas le plus courant : avec une primitive rapide mais non équitable, on peut implanter (laborieusement) une solution équitable, mais avec une primitive lente et équitable, on ne peut pas implanter une solution rapide.
2. Variante : P non bloquant ($tryP$)

$$\left\{S.val = k\right\} r \leftarrow S.tryP() \left\{ \begin{array}{c} (k > 0 \wedge S.val = k - 1 \wedge r) \\ \vee (k = 0 \wedge S.val = k \wedge \neg r) \end{array} \right\}$$

Attention aux mauvais usages : incite à l'attente active.

Sémaphore binaire – Verrou

Définition

Sémaphore S encapsulant un entier b tel que

$$\begin{array}{ccc} \{S.b = 1\} & S.P() & \{S.b = 0\} \\ \{true\} & S.V() & \{S.b = 1\} \end{array}$$

- Un sémaphore binaire est différent d'un sémaphore entier initialisé à 1.
- Souvent nommé verrou/lock
- Opérations P/V = lock/unlock ou acquire/release

Plan

1 Spécification
- Introduction
- Définition
- Modèle intuitif
- Remarques

2 Utilisation des sémaphores
- Schémas de base
- Schéma producteurs/consommateurs
- Contrôle fin de l'accès concurrent aux ressources partagées

3 Mise en œuvre des sémaphores
- Utilisation des la gestion des processus
- Sémaphore général à partir de sémaphores binaires
- L'inversion de priorité

Schémas d'utilisation essentiels (1/4)

Réalisation de l'isolation : sections critiques

- Objet partagé :

```
mutex = new Semaphore(1) // initialisé à 1
```

- Protocole d'exclusion mutuelle (pour *chacun* des processus) :

`mutex.P()` `section critique` `mutex.V()`

Généralisation :
limiter à *Max* le nombre d'utilisateurs simultanés d'une ressource *R*

- Objet partagé :

```
accèsR = new Semaphore(Max) // initialisé à Max
```

- Protocole d'accès à la ressource R (pour *chaque* processus) :

`accèsR.P()` `accès à la ressource R` `accèsR.V()`

Schémas d'utilisation essentiels (2/4)

Synchronisation élémentaire : attendre/signaler un événement E

- Objet partagé :

```
occurrenceE = new Semaphore(0) // initialisé à 0
```

- attendre une occurrence de E : `occurrenceE.P()`
- signaler l'occurence de l'événement E : `occurrenceE.V()`

Schémas d'utilisation essentiels (3/4)

Synchronisation élémentaire : rendez-vous entre 2 processus *A* et *B*

Problème : garantir l'exécution ≪ virtuellement ≫ simultanée d'un point donné du flot de contrôle de *A* et d'un point donné du flot de contrôle de *B*

- Objets partagés :

```
aArrivé = new Semaphore(0);
bArrivé = new Semaphore(0)  // initialisés à 0
```

- Protocole de rendez-vous :

```
Processus A          Processus B
  ...                  ...
aArrivé.V()          bArrivé.V()
bArrivé.P()          aArrivé.P()
  ...                  ...
```


Schémas d'utilisation essentiels (4/4)

Généralisation : rendez-vous à N processus (≪ barrière ≫)

Fonctionnement : pour passer la barrière, un processus doit attendre que les $N-1$ autres processus l'aient atteint.

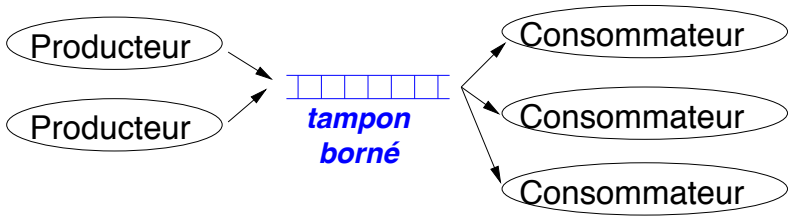
- Objet partagé :

```
barrière = tableau [0..N-1] de Semaphore;
pour i  := 0 à N-1 faire barrière[i].init(0) finpour;
```

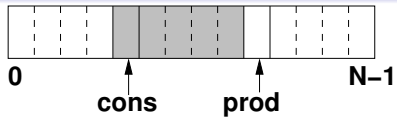
- Protocole de passage de la barrière (pour le processus *i*) :

```
pour k := 0 à N-1 faire
   barrière[i].V()
finpour;
pour k := 0 à N-1 faire
   barrière[k].P()
finpour;
```


Schéma producteurs/consommateurs : tampon borné



- tampon de taille borné et fixé
- nombre indéterminé et dynamique de producteurs
- " " " " de consommateurs



producteur

```
produire(i) {i : Item}
libre.P()
{ ∃ places libres }
mutex.P()
     { dépôt dans le tampon }
     tampon[prod] := i
     prod := prod + 1 mod N
mutex.V()
{ ∃ places occupées }
occupé.V()
```

consommateur

```
occupé.P()
{ ∃ places occupées }
mutex.P()
     { retrait du tampon }
     i := tampon[cons]
     cons := cons + 1 mod N
mutex.V()
{ ∃ places libres }
libre.V()
consommer(i) {i : Item}
```

Sémaphores : mutex := 1, occupé := 0, libre := 0 N

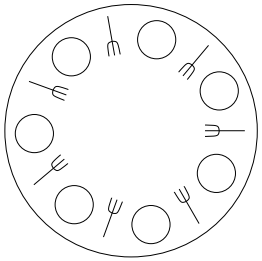
Contrôle fin du partage (1/3) : pool de ressources

- N ressources critiques, équivalentes, réutilisables
- usage exclusif des ressources
- opération `allouer` $k \leq N$ ressources
- opération `libérer` des ressources précédemment obtenues
- bon comportement :
  - pas deux demandes d'allocation consécutives sans libération intermédiaire
  - un processus ne libère pas plus que ce qu'il détient

Mise en œuvre de politiques d'allocation : FIFO, priorités...

Contrôle fin du partage (2/3) : philosophes et spaghettis

N philosophes sont autour d'une table.
Il y a une assiette par philosophe, et une fourchette entre chaque assiette.
Pour manger, un philosophe doit utiliser les deux fourchettes adjacentes à son assiette (et celles-là seulement).



Un philosophe peut être :

- penseur : il n'utilise pas de fourchettes ;
- mangeur : il utilise les deux fourchettes adjacentes ; aucun de ses voisins ne peut manger ;
- demandeur : il souhaite manger mais ne dispose pas des deux fourchettes.

Allocation multiple de ressources différenciées, interblocage...

Contrôle fin du partage (3/3) : lecteurs/rédacteurs

Une ressource peut être utilisée :

- concurremment par plusieurs lecteurs (plusieurs lecteurs simultanément) ;
- exclusivement par un rédacteur (pas d'autre rédacteur, pas d'autre lecteur).

Souvent rencontré sous la forme de verrou lecture/écriture (read-write lock).
Permet l'isolation des modifications avec un meilleur parallélisme que l'exclusion mutuelle.

Stratégies d'allocation pour des classes distinctes de clients ...

Plan

1 Spécification
- Introduction
- Définition
- Modèle intuitif
- Remarques

2 Utilisation des sémaphores
- Schémas de base
- Schéma producteurs/consommateurs
- Contrôle fin de l'accès concurrent aux ressources partagées

3 Mise en œuvre des sémaphores
- Utilisation des la gestion des processus
- Sémaphore général à partir de sémaphores binaires
- L'inversion de priorité

Implantation d'un sémaphore

Repose sur un service de gestion des processus fournissant :

- l'exclusion mutuelle (cf partie II)
- le blocage (suspension) et déblocage (reprise) des processus

Implantation

Sémaphore = ⟨ int nbjetons ;
File<Processus> bloqués ⟩

Algorithme

```
S.P() =  entrer en excl. mutuelle
           si S.nbjetons = 0 alors
              insérer self dans S.bloqués
              suspendre le processus courant
           sinon
              S.nbjetons ← S.nbjetons - 1
           finsi
         sortir d'excl. mutuelle
---------------------------------------------
S.V() =  entrer en excl. mutuelle
           si S.bloqués ≠ vide alors
              procRéveillé ← extraire de S.bloqués
              débloquer procRéveillé
           sinon
              S.nbjetons ← S.nbjetons + 1
           finsi
         sortir d'excl. mutuelle
```


Compléments (1/3) :

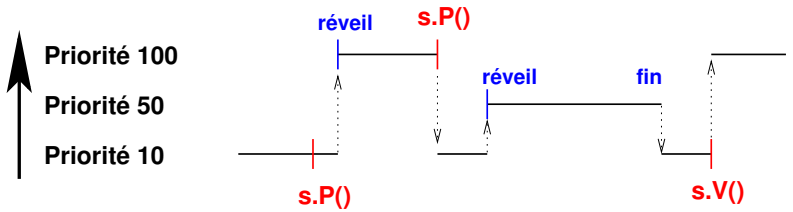
réalisation d'un sémaphore général à partir de sémaphores binaires

```
Sg = ⟨val :=?,
      mutex = new SemaphoreBinaire(1),
      accès = new SemaphoreBinaire(val>0;1;0) // verrous
      ⟩
Sg.P() =  Sg.accès.P()
          Sg.mutex.P()
          S.val ← S.val - 1
          si S.val ≥ 1 alors Sg.accès.V()
          Sg.mutex.V()
Sg.V() =  Sg.mutex.P()
          S.val ← S.val + 1
          si S.val = 1 alors Sg.accès.V()
          Sg.mutex.V()
```

→ les sémaphores binaires ont (au moins) la même puissance d'expression que les sémaphores généraux

Compléments (2/3) : sémaphores et priorités

Temps-réel ⇒ priorité ⇒ sémaphore non-FIFO.
Inversion de priorités : un processus moins prioritaire bloque/retarde indirectement un processus plus prioritaire.



Compléments (3/3) : solution à l'inversion de priorité

- Plafonnement de priorité (priority ceiling) : monter systématiquement la priorité d'un processus verrouilleur à la priorité maximale des processus potentiellement utilisateurs de cette ressource.
  - – Nécessite de connaître a priori les demandeurs
  - – Augmente la priorité même en l'absence de conflit
  - \+ Simple et facile à implanter
  - \+ Prédictible : la priorité est associée à la ressource
- Héritage de priorité : monter dynamiquement la priorité d'un processus verrouilleur à celle du demandeur.
  - \+ Limite les cas d'augmentation de priorité aux cas de conflit
  - – Nécessite de connaître les possesseurs d'un sémaphore
  - – Dynamique ⇒ comportement moins prédictible

Conclusion

Les sémaphores

- \+ ont une sémantique, un fonctionnement simples à comprendre
- \+ peuvent être mis en œuvre de manière efficace
- \+ sont suffisants pour réaliser les schémas de synchronisation nécessaires à la coordination des applications concurrentes
- – mais sont un outil de synchronisation élémentaire, aboutissant à des solutions difficiles à concevoir et à vérifier
  → schémas génériques

Quatrième partie

Interblocage

Contenu de cette partie

- définition et caractérisation des situations d'interblocage
- protocoles de traitement de l'interblocage
  - préventifs
  - curatifs
- apport déterminant d'une bonne modélisation/formalisation pour la recherche de solutions

Plan

1. L'allocation de ressources multiples
2. L'interblocage
   - Le problème
   - Condition nécessaire d'interblocage
3. Prévention
   - Approches statiques : empêcher, par construction, la formation de cycles dans le graphe d'allocation
   - Approche dynamique : esquive
4. Détection
5. Conclusion

Allocation de ressources multiples

But : gérer la compétition entre activités

- N processus, 1 ressource → protocole d'exclusion mutuelle
- N processus, M ressources → ? ? ? ?

Modèle/protocole ≪ général ≫

- Ressources banalisées, réutilisables, identifiées
- Ressources allouées par un gérant de ressources
- Interface du gérant :
  - **demander** (NbRessources) : {IdRessource}
  - **libérer** ({IdRessource})
- Le gérant :
  - rend les ressources libérées utilisables par d'autres processus
  - libère les ressources détenues, à la terminaison d'un processus.

Garanties sur les réponses aux demandes d'allocation par le gérant

- Vivacité faible (*progression*) :
  si *des* processus déposent des requêtes continûment,
  l'*une* d'entre elles finira par être satisfaite ;
- Vivacité forte (*équité faible*) :
  si un processus dépose sa requête de manière continue,
  elle finira par être satisfaite ;

Négation de la vivacité forte : famine (privation)

Un processus est en famine lorsqu'il attend infiniment longtemps la satisfaction de sa requête (elle n'est jamais satisfaite).

Plan

1. L'allocation de ressources multiples
2. L'interblocage
   - Le problème
   - Condition nécessaire d'interblocage
3. Prévention
   - Approches statiques : empêcher, par construction, la formation de cycles dans le graphe d'allocation
   - Approche dynamique : esquive
4. Détection
5. Conclusion

Le problème

Contexte : allocation de ressources réutilisables

- non réquisitionnables
- non partageables
- en quantités entières et finies
- dont l'usage est indépendant de l'ordre d'allocation

Problème

P_1 demande A puis B,
P_2 demande B puis A
$\rightarrow$ risque d'interblocage :

1. P_1 demande et obtient A
2. P_2 demande et obtient B
3. P_2 demande A
4. P_1 demande B

Interblocage : définition

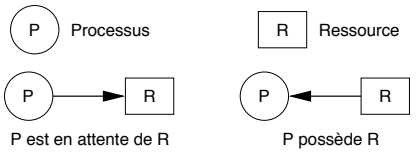
Un ensemble de processus est en interblocage si et seulement si tout processus de l'ensemble est en attente d'une ressource qui ne peut être libérée que par un autre processus de cet ensemble.

Pour l'ensemble de processus considéré :

Interblocage $\equiv$ négation de la vivacité faible (progression)

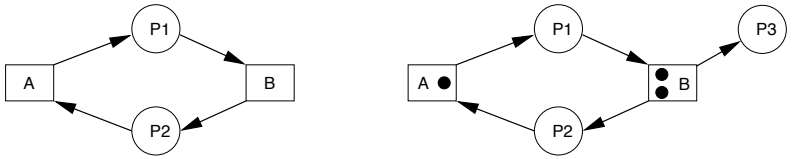
$\rightarrow$ absence de famine (viv. forte) $\Rightarrow$ absence d'interblocage (viv. faible)

Notation : graphe d'allocation



Condition nécessaire à l'interblocage

Attente circulaire (cycle dans le graphe d'allocation)



Solutions

Prévention : empêcher la formation de cycles dans le graphe

Détection + guérison : détecter l'interblocage, et l'éliminer

Plan

1 L'allocation de ressources multiples

2 L'interblocage
- Le problème
- Condition nécessaire d'interblocage

3 Prévention
- Approches statiques : empêcher, par construction, la formation de cycles dans le graphe d'allocation
- Approche dynamique : esquive

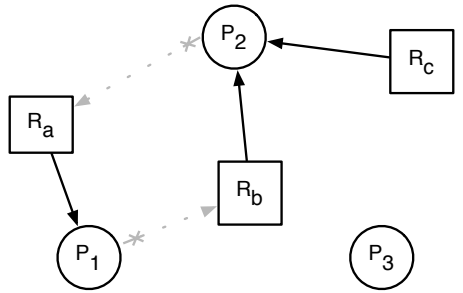
4 Détection

5 Conclusion

Comment éviter **par construction** la formation de cycles ? (1/4)

Éviter le blocage des processus

→ pas d'attente → pas d'arcs sortant d'un processus



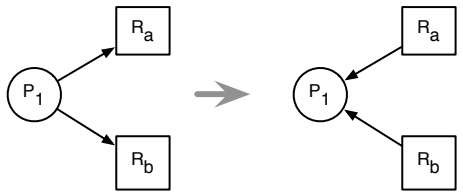
- *Ressources virtuelles* : imprimantes, fichiers
- *Acquisition non bloquante* : le demandeur peut ajuster sa demande si elle ne peut être immédiatement satisfaite

Comment éviter par construction la formation de cycles ? (2/4)

Éviter les demandes fractionnées

Allocation globale : chaque processus demande et obtient en bloc, en une seule fois, toutes les ressources nécessaires

→ une seule demande pour chaque processus
- demande satisfaite → arcs entrants uniquement
- demande non satisfaite → arcs sortants (attente) uniquement



- suppose la connaissance a priori des ressources nécessaires
- sur-allocation et risque de famine

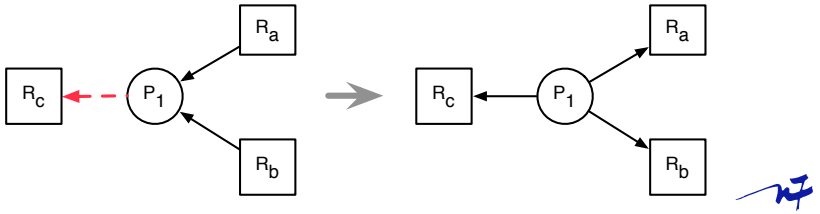
Comment éviter par construction la formation de cycles ? (3/4)

Permettre la réquisition des ressources allouées

→ éliminer/inverser les arcs entrants d'un processus en cas de création d'arcs sortants

Un processus bloqué doit

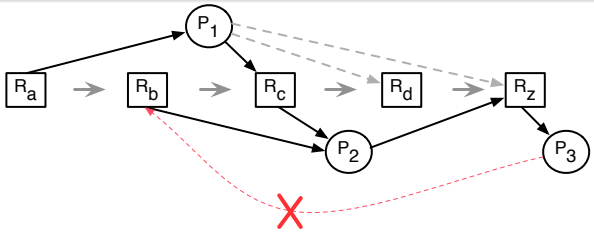
- libérer les ressources qu'il a obtenues
- réobtenir les ressources libérées, avant de pouvoir poursuivre
  → risque de famine



Comment éviter par construction la formation de cycles ? (4/4)

Fixer un ordre global sur les demandes : classes ordonnées

- un ordre est défini sur les ressources
- tout processus doit demander les ressources en suivant cet ordre



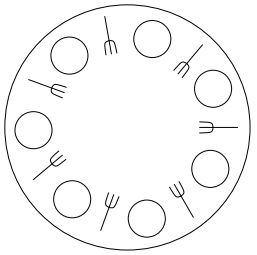
→ pour chaque processus, les chemins du graphe d'allocation vont des ressources inférieures (déjà obtenues) aux supérieures (demandées)

⇒ tout chemin du graphe d'allocation suit l'ordre des ressources

⇒ le graphe d'allocation est sans cycle
(car un cycle est un chemin sur lequel l'ordre des ressources n'est pas respecté)

Exemple : philosophes et interblocage (1/2)

N philosophes sont autour d'une table. Il y a une assiette par philosophe, et une fourchette entre chaque assiette. Pour manger, un philosophe doit utiliser les deux fourchettes adjacentes à son assiette (et celles-là seulement).



Un philosophe peut être :

- penseur : il n'utilise pas de fourchettes ;
- mangeur : il utilise les deux fourchettes adjacentes ; aucun de ses voisins ne peut manger ;
- demandeur : il souhaite manger mais ne dispose pas des deux fourchettes.

Exemple : philosophes et interblocage (2/2)

Risque d'interblocage

Chaque philosophe demande sa fourchette gauche et l'obtient. Puis quand tous ont leur fourchette gauche, chaque philosophe demande sa fourchette droite et se bloque. ⇒ interblocage

Solutions

Allocation globale : chaque philosophe demande simultanément les deux fourchettes.

Non conservation : quand un philosophe essaye de prendre sa seconde fourchette et qu'elle est déjà prise, il relâche la première et se met en attente sur la seconde.

Classes ordonnées : imposer un ordre sur les fourchettes ≡ tous les philosophes prennent d'abord la gauche puis la droite, sauf un qui prend d'abord droite puis gauche.

Esquive

Avant toute allocation, évaluation dynamique du risque (ultérieur) d'interblocage, compte tenu des ressources déjà allouées.

L'algorithme du banquier

- chaque processus annonce le nombre maximum de ressources qu'il est susceptible de demander ;
- l'algorithme maintient le système dans un état fiable, c'est-à-dire tel qu'il existe toujours une possibilité d'éviter l'interblocage dans le pire des scénarios (= celui où chaque processus demande la totalité des ressources annoncées) ;
- lorsque la requête mène à un état non fiable, elle n'est pas traitée, mais est mise en attente (comme si les ressources n'étaient pas disponibles).

Algorithme du banquier : exemple

12 ressources,
3 processus $P_0/P_1/P_2$ annonçant 10/4/9 comme maximum

	max	poss.	dem	
P_0	10	5		
P_1	4	2	+1	oui

$$(5+\mathbf{4}+2\leq 12)\wedge(\mathbf{10}+(0)+2\leq 12)\wedge((0)+(0)+\mathbf{9}\leq 12)$$

P_2	9	2	+1	non

$$(\mathbf{10}+2+3>12)\wedge(5+2+\mathbf{9}>12)\wedge(5+\mathbf{4}+3\leq 12)\wedge(\mathbf{10}+(0)+3>12)\wedge(5+(0)+\mathbf{9}>12))$$

Algorithme du banquier (1/2)

Allocation de Demande ressources au processus IdProc

```
 var Demande, Disponibles : entier = 0,N;
     Annoncées, Allouées : tableau [1..NbProc] de entier;

si Allouées[IdProc]+Demande > Annoncées[IdProc] alors erreur
sinon
  si Demande > Disponible alors <suspendre le processus>
  sinon
    si étatFiable({1..NbProc}, Disponibles - Demande) alors
      Allouées[IdProc] := Allouées[IdProc] + Demande ;
      Disponibles := Disponibles - Demande;
    sinon <suspendre le processus>;
    finsi
  finsi
finsi

fonction étatFiable(demandeurs : ensemble de 1..NbProc,
                    dispo : entier) : booléen {...}
```


Algorithme du banquier (2/2)

fonction étatFiable(demandeurs:ensemble de 1..NbProc, dispo : entier): booléen

```
 var d : 1..NbProc;
     vus, S : ensemble de 1..NbProc := ∅, ∅;
     solution : booléen := (demandeurs = ∅);

début
  répéter
   S := {p∈demandeurs-vus / Annoncées[p]-Allouées[p] <= dispo}
   si S ≠ ∅ alors
      choisir d ∈ S;
      vus := vus ∪ {d};
      solution := étatFiable(demandeurs-{d},
                             dispo+Annoncées[d]-Allouées[d]);
   finsi;
  jusqu'à (S = ∅) ou (solution);
  renvoyer solution;
fin étatFiable;
```

Plan

1. L'allocation de ressources multiples
2. L'interblocage
   - Le problème
   - Condition nécessaire d'interblocage
3. Prévention
   - Approches statiques : empêcher, par construction, la formation de cycles dans le graphe d'allocation
   - Approche dynamique : esquive
4. Détection
5. Conclusion

Détection

- construire le graphe d'allocation
- détecter l'existence d'un cycle

Coûteux → exécution périodique (et non à chaque allocation)

Guérison : Réquisition des ressources allouées à un/des processus interbloqués

- fixer des critères de choix du processus victime (priorités...)
- annulation du travail effectué par le(s) processus victime(s)
  - coûteux (détection + choix + travail perdu + restauration),
  - pas toujours acceptable (systèmes interactifs ou embarqués).

- plus de parallélisme dans l'accès aux ressources qu'avec la prévention.
- la guérison peut être un service en soi (tolérance aux pannes...)
  → Mécanismes de reprise : service de sauvegarde périodique d'états intermédiaires *(points de reprise)*

Plan

1. L'allocation de ressources multiples
2. L'interblocage
   - Le problème
   - Condition nécessaire d'interblocage
3. Prévention
   - Approches statiques : empêcher, par construction, la formation de cycles dans le graphe d'allocation
   - Approche dynamique : esquive
4. Détection
5. Conclusion

- Usuellement : interblocage = inconvénient occasionnel
  → laissé à la charge de l'utilisateur/du programmeur
  - traitement :
    - utilisation de méthodes de prévention simples (classes ordonnées, par exemple)
    - ou détection empirique (délai de garde) et guérison par choix ≪ manuel ≫ des victimes
- Cas particulier :
  systèmes ouverts, (plus ou moins) contraints par le temps
  - systèmes interactifs, multiprocesseurs, systèmes embarqués
  - recherche de méthodes efficaces, prédictibles, ou automatiques
  - compromis/choix à réaliser entre
    - la prévention qui est plus statique, coûteuse et restreint le parallélisme
    - la guérison, qui est moins prédictible, et coûteuse quand les conflits sont fréquents.
  - émergence d'approches sans blocage (→ prévention), sur les architectures multiprocesseurs (mémoire transactionnelle)

Cinquième partie

Moniteurs

Contenu de cette partie

- motivation et présentation
  d'un objet de synchronisation ≪ structuré ≫ (moniteur)
- démarche de conception basée sur l'utilisation de moniteurs
- exemple récapitulatif (schéma producteurs/consommateurs)
- annexe : variantes et mise en œuvre des moniteurs

Plan

1. Introduction
2. Définition
   - Notion de moniteur Hoare, Brinch Hansen 1973
   - Structure syntaxique d'un moniteur
   - Expression de la synchronisation : type ≪ condition ≫
   - Exemple
   - Transfert du contrôle exclusif
3. Utilisation des moniteurs
   - Méthodologie
   - Exemple : producteurs/consommateurs
4. Conclusion
5. Annexes
   - Régions critiques
   - Implémentation des moniteurs par des sémaphores FIFO

Limites des sémaphores

- imbrication aspects de synchronisation/aspects fonctionnels
  → manque de modularité, code des processus interdépendant
- pas de contrainte sur le protocole d'utilisation des sémaphores
  → démarche de conception artisanale, à partir de schémas élémentaires (attendre/signaler un événement, contrôler l'accès à une ressource...)
- approche (→ raisonnement) *opératoire*
  → vérification difficile

Exemples

- sections critiques entrelacées → interblocage
- attente infinie en entrée d'une section critique

Plan

1. Introduction
2. Définition
   - Notion de moniteur Hoare, Brinch Hansen 1973
   - Structure syntaxique d'un moniteur
   - Expression de la synchronisation : type « condition »
   - Exemple
   - Transfert du contrôle exclusif
3. Utilisation des moniteurs
   - Méthodologie
   - Exemple : producteurs/consommateurs
4. Conclusion
5. Annexes
   - Régions critiques
   - Implémentation des moniteurs par des sémaphores FIFO

Notion de moniteur Hoare, Brinch-Hansen 1973

Idée de base

La synchronisation résulte du besoin de partager «convenablement» un objet entre plusieurs processus concurrents

→ un moniteur est une construction qui permet de définir et de contrôler le bon usage d'un objet partagé par un ensemble de processus

Définition

Un moniteur = un module exportant des procédures (*opérations*)

- Contrainte : exécution des procédures du moniteur en exclusion mutuelle
- La synchronisation des opérations du moniteur est réalisée par des opérateurs internes au moniteur.

Un moniteur est passif : ce sont les processus utilisant le moniteur qui l'activent, en invoquant ses procédures.

Structure syntaxique d'un moniteur

```
<nom du moniteur> moniteur
  var <déclaration des variables partagées> ;

{opérations appelables par les processus utilisateurs}
  entrée op1(paramètres)
    var <déclaration des variables locales à op1> ;
    <code de op1>
  fin_entrée ;
  ⋮

{ procédures privées du moniteur }
  procédure x(paramètres)
    <variables locales et code de x>
  fin_ procédure ;
  ⋮

  initialisation
    <code de l'initialisation>
  fin _initialisation;
fin_moniteur;
```

Expression de la synchronisation : type *condition*

La synchronisation est définie *au sein du moniteur*, en utilisant des variables de type *condition*, internes au moniteur

- Une file d'attente est associée à *chaque* variable condition
- Opérations possibles sur une variable de type condition C :
  - C.attendre() : bloque et range dans la file associée à C le processus appelant, puis libère l'accès exclusif au moniteur.
  - C.signaler() : si des processus sont bloqués sur C, en réveille un ; sinon, nop (opération nulle).

- condition ≈ événement
  - → condition ≠ sémaphore (pas de mémorisation des « signaux »)
  - → condition ≠ prédicat logique
- Terminologie : *attendre* ↔ *wait* ; *signaler* ↔ *signal*
- Autres opérations sur les conditions :
  - C.vide() : renvoie vrai si aucun processus n'est bloqué sur C
  - C.attendre(priorité) : réveil des processus bloqués sur C selon une priorité donnée

Exemple : travail délégué (schéma client/serveur asynchrone) : 1 client + 1 serveur

Les activités (processus utilisant le moniteur)

Client	Serveur
boucle	***boucle***
⋮	⋮
déposer_travail(t)	x ← prendre_travail()
⋮	// (y ← f(x))
r ←lire_résultat()	rendre_résultat(y)
⋮	⋮
fin_boucle	***fin_boucle***

Exemple – le moniteur

Le moniteur

variables d'état : Req, Rés --Requête/Résultat en attente (null si aucun(e))
variables condition : Dépôt, Dispo

entrée déposer_travail(***in*** t)	***entrée*** prendre_travail(***out*** t)
{(pas d'attente)}	***si*** Req = null ***alors***
Req ← t	Dépôt.attendre()
Dépôt.signaler()	***finsi***
	t ← Req
	Req ← null
	{RAS}
entrée lire_résultat(***out*** r)	***entrée*** rendre_résultat(***in*** y)
si Rés = null ***alors***	
Dispo.attendre()	{(pas d'attente)}
finsi	
r ← Rés	Rés ← y
Rés ← null	
{RAS}	Dispo.signaler()

Transfert du contrôle exclusif

Les opérations du moniteur s'exécutent en exclusion mutuelle.
→ Lors d'un réveil par ***signaler()***, qui obtient l'accès exclusif ?

Priorité au signalé

Lors du réveil par ***signaler()***,

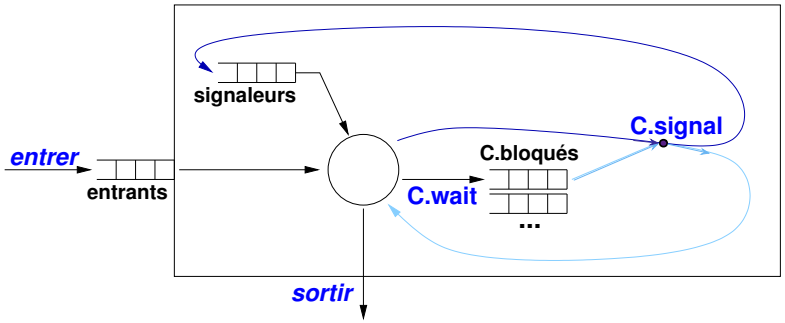
- l'accès exclusif est transféré au processus réveillé (signalé) ;
- le processus signaleur est mis en attente dans une file globale spécifique, prioritaire sur les processus entrants

Priorité au signaleur

Lors du réveil par ***signaler()***,

- l'accès exclusif est conservé par le processus réveilleur ;
- le processus réveillé (signalé) est mis en attente
  - soit dans une file globale spécifique, prioritaire sur les processus entrants,
  - soit avec les processus entrants.

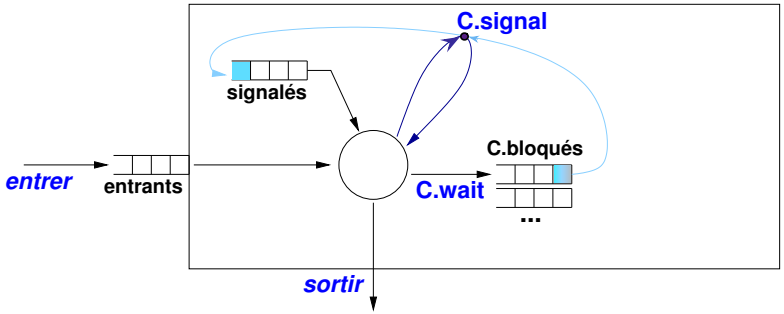
Priorité au signalé



C.*signal*()

- = opération nulle si pas de bloqués sur *C*
- sinon,
  - suspend et ajoute le signaleur à la file des signaleurs
  - extrait le processus en tête des bloqués sur *C* et lui passe le contrôle
- signaleurs prioritaires sur les entrants (progression garantie)

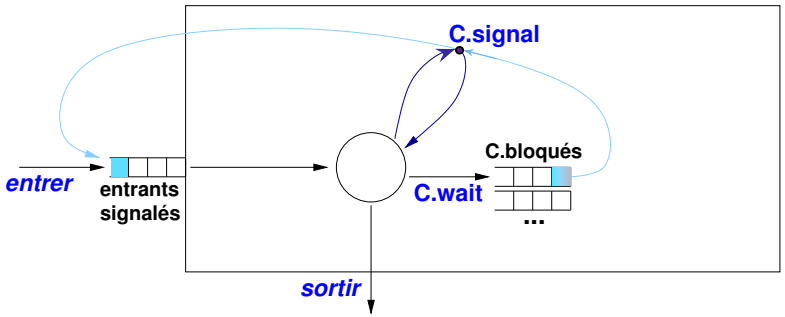
Priorité au signaleur avec file spécifique des signalés



C.*signal*()

- si la file des bloqués sur *C* est non vide, en extrait le processus de tête et le range dans la file des signalés
- le signaleur conserve le contrôle
- signalés prioritaires sur les entrants

Priorité au signaleur sans file spécifique des signalés



C.*signal*()

- si la file des bloqués sur *C* est non vide, en extrait le processus de tête et le range dans la file des entrants
- le signaleur conserve le contrôle
- signalés non prioritaires vis-à-vis des entrants

Comparaison des stratégies de transfert du contrôle

- **Priorité au signalé** : garantit que le processus réveillé obtient l'accès au moniteur dans l'état où il était lors du signal.
  - Raisonnement simplifié (le signaleur produit un état, directement utilisé par le signalé)
  - Absence de famine facilitée
- **Priorité au signaleur** : le réveillé obtient le moniteur ultérieurement, éventuellement après d'autres processus
  - Implantation du mécanisme plus simple et plus performante
  - Au réveil, le signalé doit retester la condition de déblocage
    - → Possibilité de famine, écriture et raisonnements plus lourds

Peut-on simplifier encore l'expression de la synchronisation ?

Idée (d'origine)

Attente sur des prédicats,

plutôt que sur des événements (= variables de type condition)

→ opération unique : *attendre(B)*, B expression booléenne

Exemple : moniteur pour le tampon borné, avec *attendre*(prédicat)

```
variables d'état : Req, Rés --Requête/Résultat en attente (null si aucun(e))
```

entrée déposer_travail(**in** t)
  Req ← t
	entrée prendre_travail(**out** t)
	  attendre(Req ≠ null)
	  t ← Req
	  Req ← null

entrée lire_résultat(**out** r)
  attendre(Rés ≠ null)
  r ← Rés
  Rés ← null
	entrée rendre_résultat(**in** y)
	  Rés ← y

Pourquoi *attendre*(prédicat) n'est-elle pas disponible en pratique ?

Efficacité problématique :

⇒ évaluer B à chaque nouvel état (= à chaque affectation), et pour chacun des prédicats attendus.

→ **gestion de l'évaluation laissée au programmeur**

- à chaque prédicat attendu (P) est associée une variable de type condition (P_valide)
- *attendre*(P) est implantée par **si** ¬ P **alors** P_valide.*attendre*() **fsi** {P}
- le programmeur a la possibilité de signaler ($P\_valide.signaler()$) les instants/états (pertinents) où P est valide

Principe

- concevoir en termes de prédicats attendus, puis
- simuler cette attente de prédicats au moyen de variables de type condition

Plan

1. Introduction
2. Définition
   - Notion de moniteur Hoare, Brinch Hansen 1973
   - Structure syntaxique d'un moniteur
   - Expression de la synchronisation : type « condition »
   - Exemple
   - Transfert du contrôle exclusif
3. Utilisation des moniteurs
   - Méthodologie
   - Exemple : producteurs/consommateurs
4. Conclusion
5. Annexes
   - Régions critiques
   - Implémentation des moniteurs par des sémaphores FIFO

Méthodologie (1/3)

Motivation

Moniteur = réalisation (et gestion) d'un objet partagé

→ permet de concevoir la synchronisation en termes d'interactions entre chaque processus et un objet partagé : les seules interactions autorisées sont celles qui laissent l'objet partagé dans un état cohérent

→ Invariant du moniteur = ensemble des états possibles pour l'objet géré par le moniteur

Protocole générique : exécution d'une action A sur un objet partagé, caractérisé par un invariant I

1. ***si*** l'exécution de A (depuis l'état courant) invalide I ***alors attendre() finsi*** { prédicat d'acceptation de A}
2. Effectuer A { → nouvel état courant E }
3. ***Réveiller()*** les processus en attente qui peuvent effectuer des actions à partir de E

Méthodologie (2/3)

Etapes

1. Déterminer l'interface du moniteur
2. Énoncer en français les prédicats d'acceptation de chaque opération
3. Déduire les variables d'état qui permettent d'écrire ces prédicats d'acceptation
4. Formuler l'invariant du moniteur et les prédicats d'acceptation
5. Associer à chaque prédicat d'acceptation une variable condition qui permettra d'attendre/signaler la validité du prédicat
6. Programmer les opérations, en suivant le protocole générique précédent
7. Vérifier que
   - l'invariant est vrai chaque fois que le contrôle du moniteur est transféré
   - les réveils ont lieu quand le prédicat d'acceptation est vrai

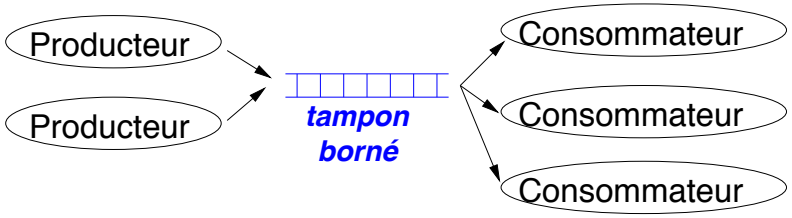
Méthodologie (3/3)

Structure standard d'une opération

si le prédicat d'acceptation est faux ***alors***
  attendre() sur la variable condition associée
finsi
{ (1) État nécessaire au bon déroulement }
Mise à jour de l'état du moniteur (action)
{ (2) État garanti (résultat de l'action) }
signaler() les variables conditions dont le prédicat associé est vrai

Vérifier, pour chaque variable condition, que chaque précondition de ***signaler()*** (2) implique chaque postcondition de ***attendre()*** (1)

Exemple : réalisation du schéma producteurs/consommateurs



- tampon de taille borné et fixé
- nombre indéterminé et dynamique de producteurs
- " " " " de consommateurs

1. Interface :
   - `déposer(in v)`
   - `retirer(out v)`
2. Prédicats d'acceptation :
   - déposer : il y a de la place, le tampon n'est pas plein
   - retirer : il y a quelque chose, le tampon n'est pas vide
3. Variables d'état :
   - `nbOccupées` : natural
   - déposer : `nbOccupées` $< N$
   - retirer : `nbOccupées` > 0
4. Invariant : $0 \leq$ `nbOccupées` $\leq N$
5. Variables conditions : `PasPlein`, `PasVide`

déposer(in v)

si $\neg$(`nbOccupées` $< N$) ***alors***
  `PasPlein.`***attendre()***
finsi
{ `nbOccupées` $< N$ }
`// action applicative (ranger v dans le tampon)`
`nbOccupées` $++$
{ $N \geq$ `nbOccupées` > 0 }
`PasVide.`***signaler()***

retirer(out v)

si $\neg$(`nbOccupées` > 0) ***alors***
  `PasVide.`***attendre()***
finsi
{ `nbOccupées` > 0 }
`// action applicative (prendre v dans le tampon)`
`nbOccupées` $--$
{ $0 \leq$ `nbOccupées` $< N$ }
`PasPlein.`***signaler()***

Plan

1. Introduction
2. Définition
   - Notion de moniteur Hoare, Brinch Hansen 1973
   - Structure syntaxique d'un moniteur
   - Expression de la synchronisation : type « condition »
   - Exemple
   - Transfert du contrôle exclusif
3. Utilisation des moniteurs
   - Méthodologie
   - Exemple : producteurs/consommateurs
4. Conclusion
5. Annexes
   - Régions critiques
   - Implémentation des moniteurs par des sémaphores FIFO

Plan

1. Introduction
2. Définition
   - Notion de moniteur Hoare, Brinch Hansen 1973
   - Structure syntaxique d'un moniteur
   - Expression de la synchronisation : type « condition »
   - Exemple
   - Transfert du contrôle exclusif
3. Utilisation des moniteurs
   - Méthodologie
   - Exemple : producteurs/consommateurs
4. Conclusion
5. Annexes
   - Régions critiques
   - Implémentation des moniteurs par des sémaphores FIFO

Conclusion

Un moniteur implante un objet partagé, et contrôle la bonne utilisation de cet objet

Apports
- modularité et encapsulation.
- la synchronisation est localisée dans le moniteur →
  - raisonnement simplifié
  - meilleure lisibilité

Limites
- dans le moniteur, la synchronisation reste mêlée aux aspects fonctionnels
- la sémantique des moniteurs est complexe
- l'exclusion mutuelle sur les opérations d'un moniteur facilite la conception, mais :
  - est une source potentielle d'interblocages (moniteurs imbriqués)
  - est une limite du point de vue de l'efficacité

Régions critiques

- Éliminer les variables conditions et les appels explicites à `signaler` ⇒ déblocages calculés par le système.
- Exclusion mutuelle plus « fine », en listant les variables partagées effectivement utilisées.

```
region liste des variables utilisées
when prédicat logique
do code
```

1. Attente que le prédicat logique soit vrai
2. Le code est exécuté en exclusion mutuelle vis-à-vis des autres régions ayant (au moins) une variable commune
3. À la fin du code, évaluation automatique des prédicats logiques des régions pour débloquer éventuellement.

Exemple

```
tampon : shared array 0..N-1 of msg;
nbOcc : shared int := 0;
retrait, dépôt : shared int := 0, 0;

déposer(m)                    | retirer()
  region                      |   region
    nbOcc, tampon, dépôt      |     nbOcc, tampon, retrait
  when                        |   when
    nbOcc < N                 |     nbOcc > 0
  do                          |   do
    tampon[dépôt] ← m         |     Result ← tampon[retrait]
    dépôt ← dépôt + 1 % N     |     retrait ← retrait + 1 % N
    nbOcc ← nbOcc + 1         |     nbOcc ← nbOcc - 1
  end                         |   end
```

Implémentation des moniteurs par des sémaphores FIFO

Dans le cas où les ***signaler()*** sont toujours en fin d'opération

- Exclusion mutuelle sur l'exécution des opérations du moniteur
  - définir un sémaphore d'exclusion mutuelle : ***mutex***
  - encadrer chaque opération par ***mutex.P()*** et ***mutex.V()***
- Réalisation de la synchronisation par variables condition
  - définir un sémaphore ***SemC*** (initialisé à 0) pour chaque condition ***C***
  - traduire ***C.attendre()*** par ***SemC.P()***, et ***C.signaler()*** par ***SemC.V()***
  - Difficulté : pas de mémoire pour les appels à ***C.signaler()***
    - → éviter d'exécuter ***SemC.V()*** si aucun processus n'attend
    - → un compteur explicite par condition : ***cptC***
      - Réalisation de ***C.signaler()*** :
        si cptC $>$ 0 **alors** SemC.V() **sinon** mutex.V() **fsi**
      - Réalisation de ***C.attendre()*** :
        cptC ++ ; mutex.V() ; SemC.P() ; cptC - - ;

Dans le cas général : ajout d'un compteur et d'un sémaphore pour les processus signaleurs, réveillé prioritairement par rapport à ***mutex***

Sixième partie

Programmation multi-activités

Contenu de cette partie

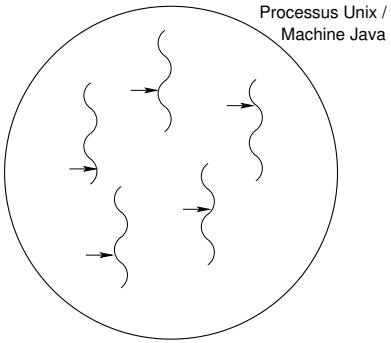
Préparation aux TPs : présentation des outils de programmation concurrente autour de la plateforme Java

- notion de processus léger
- présentation de la plateforme
- classe `Thread`
- objets de synchronisation : moniteurs, sémaphores...
- régulation des activités : pools d'activités, appels asynchrones, fork/join...
- outils de synchronisation de bas niveau
- autres environnements et modèles : Posix, OpenMP...

Plan

1. Retour sur les processus
2. Threads Java
   - Création d'une activité
   - Quelques méthodes
   - Interruption
   - Variables localisées
3. Synchronisation Java
   - Moniteurs
   - Objets de synchronisation
   - Services de régulation du parallélisme
   - Synchronisation de bas niveau/élémentaire
4. Autres environnements
5. Annexe : Threads POSIX

Processus multi-activités

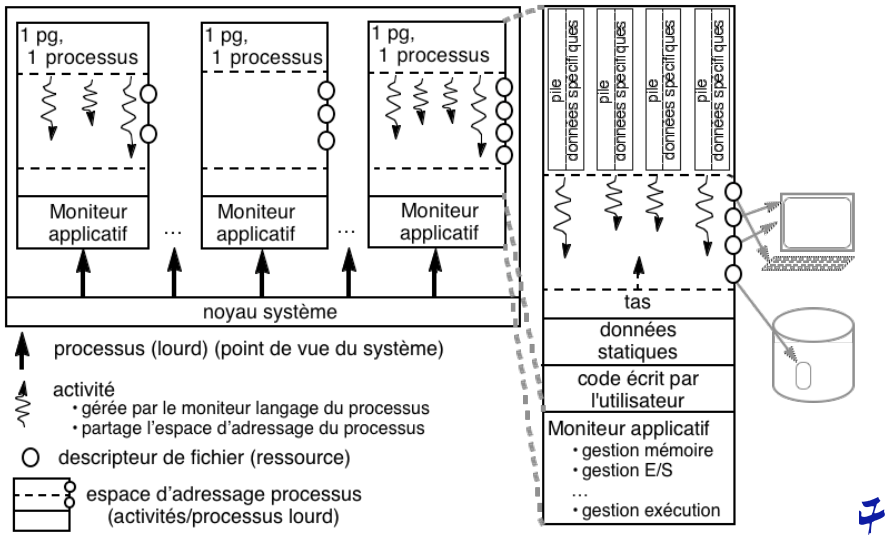


1 espace d'adressage, plusieurs flots de contrôle.
⇒ plusieurs activités (ou processus légers) au sein d'un même processus UNIX / d'une même machine virtuelle Java.

Relation et différences entre processus lourds et légers

- *Processus lourds* : représentent l'exécution d'une application, du point de vue du système
  - → unité d'allocation de ressources
    - espaces d'adressage et ressources distinctes (pas de partage)
    - commutation coûteuse (appels systèmes → passage par le mode superviseur)
- *Processus légers* (threads, activités...) :
  - unité d'exécution : résulte de la décomposition (fonctionnelle) d'un traitement en sous-traitements parallèles, pour tirer profit de la puissance de calcul disponible, ou simplifier la conception
  - les ressources (mémoire, fichiers...) du processus lourd exécutant un traitement sont partagées entre les activités réalisant ce traitement
    - chaque activité a sa pile d'exécution et son contexte processeur, mais les autres éléments sont partagés
  - une bibliothèque applicative (≪ moniteur ≫) gère le partage entre activités du temps processeur alloué au processus lourd → commutation plus efficace.

Mise en œuvre des processus légers



Difficultés de mise en œuvre des processus légers

L'activité du moniteur applicatif est opaque pour le système d'exploitation : le moniteur du langage multiplexe les ressources d'un processus lourd entre ses activités, sans appel au noyau.

→ commutation de contexte plus légère, mais
- appels système usuellement bloquants
  → 1 activité bloquée ⇒ toutes les activités bloquées
  - → utiliser des appels systèmes non bloquants (s'ils existent) au niveau du moniteur applicatif, et gérer l'attente,
- réaction aux événements asynchrones a priori ≪ lente ≫
  - → définir 1 service d'événements au niveau du moniteur applicatif, et utiliser (si c'est possible) le service d'événements système

Remarque : la mise en œuvre des processus légers est directe lorsque le système d'exploitation fournit un service d'activités noyau et permet de coupler activités noyau et activités applicatives

Conception d'applications parallèles en Java

Java permet de manipuler
- les processus (lourds) : classes `java.lang.ProcessBuilder` et `java.lang.Process`
- les activités (processus légers) : classe `java.lang.Thread`

Le degré de parallélisme des applications Java peut être
- contrôlé directement (manipulation des threads)
- ou régulé
  - explicitement : interface `java.util.concurrent.Executor`
  - implicitement : environnement proposé par Java 8 pour la programmation asynchrone/fonctionnelle/événementielle (abordé plus tard)

Plan

1. Retour sur les processus
2. Threads Java
   - Création d'une activité
   - Quelques méthodes
   - Interruption
   - Variables localisées
3. Synchronisation Java
   - Moniteurs
   - Objets de synchronisation
   - Services de régulation du parallélisme
   - Synchronisation de bas niveau/élémentaire
4. Autres environnements
5. Annexe : Threads POSIX

Création d'activités : exemple

```
class Compteur implements Runnable {
    private int max;
    private int step;
    public Compteur(int max, int step) {
        this.max = max; this.step = step;
    }
    public void run() {
        for (int i = 0; i < max; i += step)
            System.out.println(i);
    }
}
public class DemoThread {
    public static void main (String[] a) {
        Compteur c2 = new Compteur(10, 2);
        Compteur c3 = new Compteur(15, 3);
        new Thread(c2).start();
        new Thread(c3).start();
    }
}
```


Création d'une activité (Thread)

1/Définir une classe implantant l'interface Runnable (méthode run)

```
class X implements Runnable {
     public void run() { /* code du thread */ }
}
```

2/Utiliser 1 instance de cette classe Runnable pour créer un Thread

```
X x = new X(...);
Thread t = new Thread(x);   // activité créée
t.start();                  // activité démarrée
⋮
t.join();                   //attente de la terminaison (si besoin)
```

Remarque : il est aussi possible de créer une activité par héritage de la classe Thread et implantation de la méthode run

Quelques méthodes

Classe Thread :

`static Thread currentThread()`
: fournit (la référence à) l'activité appelante

`void join() throws InterruptedException`
: suspend l'exécution de l'activité appelante jusqu'à la terminaison de l'activité sur laquelle `join()` est appliquée (ou jusqu'à ce que l'activité appelante soit interrompue)

`static void sleep(long ms) throws InterruptedException`
: suspend l'exécution de l'activité appelante pendant la durée indiquée (ou jusqu'à ce que l'activité soit interrompue)

Complément : interruption

Mécanisme élémentaire permettant de signaler un événement asynchrone

- La méthode `interrupt` (appliquée à une activité) positionne un indicateur `interrupted`, testable par :
  `boolean isInterrupted()` qui renvoie la valeur de l'indicateur `interrupted` de l'activité sur laquelle cette méthode est appliquée ;
  `static boolean interrupted()` qui renvoie *et efface* la valeur de l'indicateur de l'activité appelante.
- Si l'activité sur laquelle s'applique `interrupt` est bloquée sur une opération de synchronisation qui lève l'exception `InterruptedException` (`Thread.join`, `Thread.sleep`, `Object.wait`...), celle-ci est levée, et `interrupted` est effacé.

Pas d'interruption des entrées-sorties bloquantes → intérêt limité.

Complément : variables localisées

Permet de définir un contexte d'exécution local, non partagé : chaque activité possède sa propre valeur associée à un objet localisé (qui est instance de `ThreadLocal` ou `InheritableThreadLocal`).

```
class Common {
   static ThreadLocal val = new ThreadLocal(); //attribut localisé par thread
   static Integer v = new Integer(0);          // attribut global ''standard''}
```

thread t1 : incrémente v et val

```
Integer o =  new Integer(0);
Integer x = new Integer(0);
Integer y = new Integer(1);
...
Common.val.set(o);
for (int i = 0; i <= 2; i++){
x = (Integer) Common.val.get();
o = Integer.valueOf(x.intValue()+1);
Common.val.set(o);
y = Common.v;
Common.v = Integer.valueOf(y.intValue()+1);
System.out.println("T1 - G: "+y+" / TL: "+x);}
```

thread t2 : incrémente v, autoconcatène val

```
String o = "bip ";
String x ;
Integer y = new Integer(1);
...
Common.val.set(o);
for (int i = 0; i <= 2; i++) {
x = (String) Common.val.get();
o=o+x;
Common.val.set(o);
y = Common.v;
Common.v = Integer.valueOf(y.intValue()+1);
System.out.println("T2 - G: "+y+" / TL: "+x);}
```

```
Résultat :
T1 - G: 0 / TL: 0
T2 - G: 1 / TL: bip
T1 - G: 2 / TL: 1
T2 - G: 3 / TL: bip bip
T1 - G: 4 / TL: 2
T2 - G: 5 / TL: bip bip bip bip
```

Plan

1. Retour sur les processus
2. Threads Java
   - Création d'une activité
   - Quelques méthodes
   - Interruption
   - Variables localisées
3. Synchronisation Java
   - Moniteurs
   - Objets de synchronisation
   - Services de régulation du parallélisme
   - Synchronisation de bas niveau/élémentaire
4. Autres environnements
5. Annexe : Threads POSIX

Le paquetage `java.util.concurrent` fournit

- une réalisation des moniteurs
- divers autres objets de synchronisation
  - barrière
  - sémaphore
  - compteur
  - ...
- le contrôle du degré de parallélisme : `Thread`, `Executor`
- des structures de données permettant des accès concurrents (collections « concurrentes ») de manière transparente
  - accès atomiques : `ConcurrentHashMap`...
  - accès non bloquants : `ConcurrentLinkedQueue`

Moniteur Java (5)

- un verrou assurant l'exclusion mutuelle (équité possible)
- variables conditions associées à ce verrou
- pas de priorité au signalé et pas de file des signalés

```
import java.util.concurrent.locks.*;
class ProdCon {
  Lock moniteur = new ReentrantLock();
  Condition pasPlein  = moniteur.newCondition();
  Condition pasVide = moniteur.newCondition();
  Object[] items = new Object[100];
  int depot, retrait, nbElems;

  public void deposer(Object x) throws InterruptedException {
        moniteur.lock();
           while (nbElems == items.length) pasPlein.await();
          items[depot] = x; depot = (depot + 1) % items.length;
          nbElems++;
          pasVide.signal();
        moniteur.unlock();
    }
     ⋮
```


Objets de synchronisation (1/3)

Sémaphore

```
Semaphore s = new Semaphore(1); // nb init. de jetons
s.acquire();                    // = P
s.release();                    // = V
```

BlockingQueue

`BlockingQueue` = producteurs/consommateurs (Interface)

`LinkedBlockingQueue` = prod./cons. à tampon non borné

`ArrayBlockingQueue` = prod./cons. à tampon borné

```
BlockingQueue bq;
bq.put(m);      // dépôt (bloquant) d'un objet en queue
x = bq.take(); // obtention (bloquante) de l'objet en tête
```


Objets de synchronisation (2/3)

CyclicBarrier

Rendez-vous bloquant entre N activités : passage bloquant tant que les N activités n'ont pas demandé à franchir la barrière ; passage autorisé pour toutes quand la N-ième arrive.

```
CyclicBarrier barriere = new CyclicBarrier(3);
for (int i = 0; i < 8; i++) {
    Thread t = new Thread() {
        public void run() {
            barriere.await();
            System.out.println("Passé !");
        }
    };
    t.start();
}
```

Généralisation : la classe `Phaser` permet un rendez-vous (bloquant ou non) pour un *groupe variable* d'activités.

Objets de synchronisation (3/3)

countDownLatch

`init(N)` valeur initiale du compteur

`await()` bloque si strictement positif, rien sinon.

`countDown()` décrémente (si strictement positif). Lorsque le compteur devient nul, toutes les activités bloquées sont débloquées.

interface locks.ReadWriteLock

Fournit des verrous pouvant être acquis en mode

- exclusif (méthode `writeLock()`),
- ou partagé avec les autres non exclusifs (méthode `readLock()`)

→ mise en œuvre du schéma lecteurs/rédacteurs.

→ implémentation : `ReentrantReadWriteLock` (avec/sans équité)

Services de régulation du parallélisme : exécuteurs

Idée

Séparer la création et la gestion des activités des autres aspects (fonctionnels, synchronisation...)

→ définition d'un service de gestion des activités (exécuteur), régulant/adaptant le nombre de threads effectivement actifs, en fonction de la charge courante et du nombre de processeurs physiques disponibles :

- trop de threads → consommation de ressources inutile
- pas assez de threads → capacité de calcul sous-utilisée

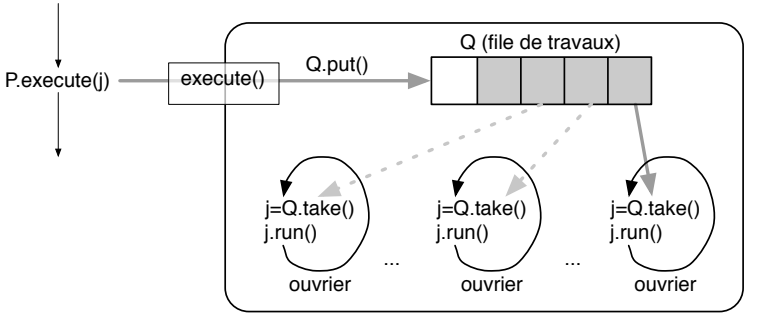
Interfaces d'exécuteurs

- l'interface `Executor` définit la méthode `execute(Runnable r)`,
  - fonctionnellement équivalente à `(new Thread(r)).start()`,
  - avec la différence que `r` ne sera pas forcément exécutée immédiatement/par un thread spécifique.
- la sous-interface `ExecutorService` permet de soumettre (méthode `submit(...)`) des tâches rendant un résultat (`Callable`), lequel pourra être récupéré par la suite, de manière asynchrone.
- l'interface `ScheduledExecutorService` est un `ExecutorService`, avec la possibilité de spécifier un calendrier (départs, périodicité...) pour les tâches exécutées.

Pools de Threads

Schéma de base pour la plupart des implémentations d'exécuteurs

- Une file d'attente de travaux à effectuer
- Un ensemble (fixe ou dynamique) d'activités (ouvriers)
- Une politique de distribution des travaux aux activités (réalisée par un protocole ou par une activité)



Pool P *[sans politique de distribution particulière (file partagée)]*

Principe de fonctionnement : pool de threads « minimal »

```
import java.util.concurrent.*;
public class PlainThreadPool {
  private BlockingQueue<Runnable> queue;
  public PlainThreadPool(int nthr) {
    queue = new LinkedBlockingQueue<Runnable>();
    for (int i=0; i<nthr; i++) { (new Ouvrier()).start(); }
  }
  public void execute(Runnable job) {
    queue.put(job);
  }
  private class Ouvrier extends Thread {
    public void run() {
      while (true) {
        Runnable job = queue.take(); //bloque si nécessaire
        job.run();
      }
    }
  }
}
```

Exécuteurs implantés par des pools de threads prédéfinis

La classe `java.util.concurrent.Executors` est une fabrique pour des stratégies d'exécution classiques :

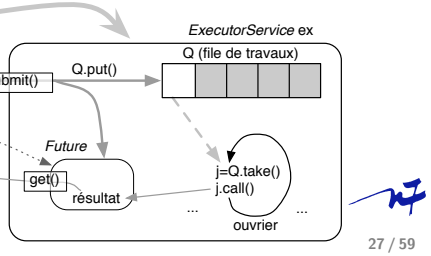
- Nombre fixe d'activités : méthodes `newSingleThreadExecutor()`, `newFixedThreadPool(int nThreads)`
- Nombre d'activités adaptable : méthode `newCachedThreadPool()`
  - Quand il n'y a plus d'activité disponible et qu'un travail est déposé, création d'une nouvelle activité
  - Quand la queue est vide et qu'un délai suffisant (p.ex. 1mn) s'est écoulé, terminaison d'une activité inoccupée
  - Possibilité de définir un calendrier pour la création/activation des threads du pool

La classe `java.util.concurrent.ThreadPoolExecutor` permet de contrôler l'ensemble des paramètres de la stratégie d'exécution : politique de la file (FIFO, priorités...), file bornée ou non bornée, nombre de threads minimum, maximum...

Évaluation asynchrone : `Callable` et `Future`

- Evaluation paresseuse : l'appel effectif d'une fonction peut être différé (éventuellement exécuté en parallèle avec l'appelant)
- `submit(...)` fournit à l'appelant une référence à la valeur future du résultat.
- L'appelant ne se bloque que quand il doit utiliser le résultat de l'appel (si l'évaluation de celui-ci n'est pas terminée).
  → appel de la méthode `get()` sur le `Future`

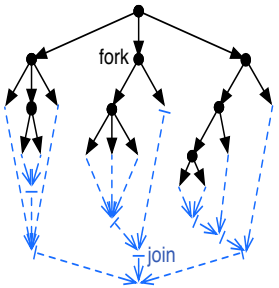
```
class FonctionAsynchrone implements Callable<TypeRetour> {
    public TypeRetour call() { ... return v; }
}
ExecutorService ex = Executors.newCachedThreadPool();
Callable<TypeRetour> fonc = new FonctionAsynchrone();
...
Future<TypeRetour> appel = ex.submit(fonc);
...
TypeRetour ret = appel.get(); //éventuellement bloquant
```



Un peu de Big Data : schéma diviser pour régner (fork/join, map/reduce)

Schéma de base

Résultat résoudre(Problème pb) {
 si (pb est assez petit) {
  résoudre directement pb
 } **sinon** {
  décomposer le problème en parties indépendantes
  fork : créer des (sous-)tâches
   pour résoudre chaque partie
  join : attendre la réalisation de ces (sous-)tâches
  fusionner les résultats partiels
  retourner le Résultat
 }

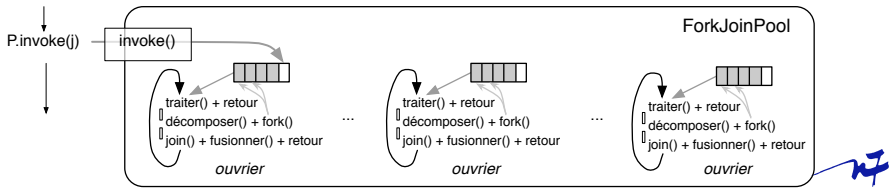


Exécuteur pour le schéma fork/join (1/3)

Difficulté de la stratégie diviser pour régner :
schéma exponentiel + coût de la création d'activités

Classe `ForkJoinPool`

- Ensemble prédéterminé (pool) d'activités, chacune équipée d'une file d'attente de travaux à traiter.
- Les activités gérées sont des instances de `ForkJoinTask` (méthodes `fork()` et `join()`)



Exécuteur pour le schéma fork/join (2/3)

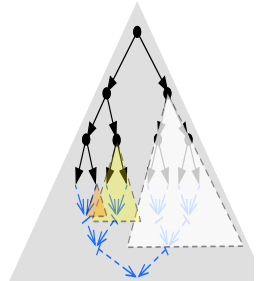
Activité d'un ouvrier du ForkJoinPool

- Un ouvrier traite la tâche placée en tête de sa file
- Un ouvrier appelant `fork()` ajoute les travaux créés en tête de sa propre file

→

Chaque ouvrier traite un arbre de tâches qu'il

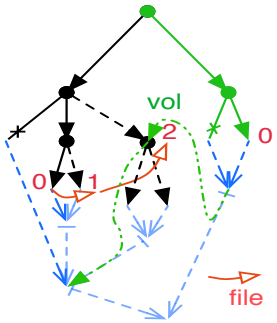
- parcourt (et traite) en profondeur d'abord (en préordre) → économie d'espace
- construit progressivement en largeur, au fur et à mesure de son parcours : lorsqu'un ouvrier descend d'un niveau, les frères de la tâche à traiter sont créés, et placés en tête de la file d'attente



Exécuteur pour le schéma fork/join (3/3)

Régulation

Vol de travail : lorsqu'une activité a épuisé les travaux de sa file, elle prend un travail en queue d'une autre file



La tâche prise correspond au dernier sous-arbre (le plus proche de la racine) qui était affecté à l'ouvrier « volé »

→ pas de conflits si les sous-problèmes sont bien partitionnés

→ pas d'attente inutile pour l'ouvrier « volé » puisque la tâche volée était la dernière à traiter.

Services de synchronisation élémentaire

Fonctions

- exclusion mutuelle
- attendre/signaler un événement

  analogue à un moniteur (priorité au signaleur, sans file des signalés) avec une seule variable condition
- accès élémentaires atomiques

Exclusion mutuelle

Toute instance d'objet Java est munie d'un verrou exclusif

Code synchronisé

```
synchronized (unObj) {
        < Région critique >
}
```

Méthode synchronisée

```
synchronized T uneMethode(...) { ... }
```

Remarques

- exclusion d'accès de l'objet sur lequel on applique la méthode, pas de la méthode elle-même
- les méthodes de classe (statiques) peuvent aussi être synchronisées : une classe est une instance de la classe `Class`.

Synchronisation associée aux verrous d'objets

Méthodes `wait` et `notify[All]` applicables à tout objet, pour lequel l'activité a obtenu l'accès exclusif.

`unObj.wait()` libère l'accès exclusif à l'objet et bloque l'activité appelante en attente d'un réveil via une opération `unObj.notify`

`unObj.notify()` réveille une seule activité bloquée sur l'objet (si aucune activité n'est bloquée, l'appel ne fait rien) ;

`unObj.notifyAll()` réveille toutes les activités bloquées sur l'objet.

Difficultés

- prises multiples de verrous :

```
synchronized(o1) { synchronized(o2) { o1.wait(); } }
```

- pas de priorité au signalé, pas d'ordonnancement sur les déblocages
- une seule notification possible pour une exclusion mutuelle donnée
  → résolution compliquée de problèmes de synchronisation
  - programmer comme avec des sémaphores
  - affecter un objet de blocage distinct à chaque requête et gérer soi-même les files d'attente

Schéma de base

```
class Requête {
    bool ok;
    // paramètres d'une demande
}
List<Requête> file;
```

demande bloquante

```
req = new Requête(...)
synchronized(file) {
   if (satisfiable(req)) {
     // + maj état applicatif
     req.ok = true;
   } else {
     file.add(req)
   }
}
synchronized(req) {
   while (! req.ok)
     req.wait();
}
```

libération

```
synchronized(file) {
   // + maj état applicatif
   for (Requête r : file) {
     synchronized(r) {
       if (satisfiable(r)) {
         // + maj état applicatif
         r.ok = true
         r.notify();
       }
     }
   }
}
```

Atomicité à grain fin

Java fournit des outils pour faciliter la conception d'algorithmes concurrents à grain fin, c'est-à-dire où la coordination sera réalisée par l'accès à des données partagées, plutôt que par suspension/réveil (attente/signal d'événement)

- les lectures et les écritures des références et de la plupart des types primitifs (`long` et `double` exceptés) sont atomiques
- idem pour les variables déclarées `volatile`
- le paquetage `java.util.concurrent.atomic` fournit des classes qui permettent des accès atomiques et offrent en outre des opérations de mise à jour conditionnelle du type `TestAndSet`

Rappel (et mise en garde)

Concevoir et valider des algorithmes de ce type est très ardu. Cette difficulté même a motivé la définition d'objets et de méthodologies de synchronisation (sémaphores,...)

Plan

1. Retour sur les processus
2. Threads Java
   - Création d'une activité
   - Quelques méthodes
   - Interruption
   - Variables localisées
3. Synchronisation Java
   - Moniteurs
   - Objets de synchronisation
   - Services de régulation du parallélisme
   - Synchronisation de bas niveau/élémentaire
4. Autres environnements
5. Annexe : Threads POSIX

Posix Threads

Standard de librairie multi-activités, supporté par de nombreuses implantations plus ou moins conformes
(SUN/Solaris 2.5, Linux, FreeBSD, HP-UX 11.0...)
Nom officiel : POSIX 1003.1-1996.
Repris dans X/Open XSH5 .
Contenu de la bibliothèque :

- manipulation d'activités (création, terminaison...)
- synchronisation : verrous, variables condition.
- primitives annexes : données spécifiques à chaque activité, politique d'ordonnancement...
- ajustement des primitives standard : processus lourd, E/S, signaux, routines réentrantes.

Windows API (C, C++)

Plus de 150 (?) fonctions, dont :

- création d'activité : `CreateThread`
- exclusion mutuelle : `InitializeCriticalSection`, `EnterCriticalSection`, `LeaveCriticalSection`
- synchronisation basique : `WaitForSingleObject`, `WaitForMultipleObjects`, `SetEvent`
- synchronisation ≪ évoluée ≫ : `SleepConditionVariableCS`, `WakeConditionVariable`

Note : l'API Posix Threads est aussi supportée.

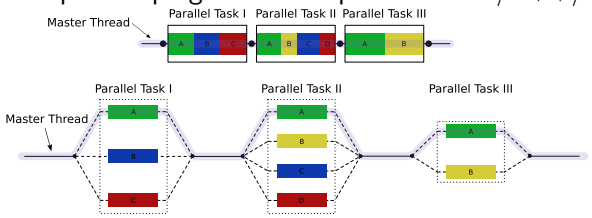
.NET (C#)

Très similaire à Java :

- Création d'activité :
  `t = new System.Threading.Thread(méthode);`
- Démarrage : `t.Start();`
- Attente de terminaison : `t.Join();`
- Exclusion mutuelle : `lock(objet) { ... }`
  (mot clef du langage)
- Synchronisation élémentaire :
  `System.Threading.Monitor.Wait(objet);`
  `System.Threading.Monitor.Pulse(objet);` (= notify)
- Sémaphore :
  `s = new System.Threading.Semaphore(nbinit,nbmax);`
  `s.Release(); s.WaitOne();`

OpenMP

- API pour la programmation parallèle en C/C++/Fortran



- Annotations dans le code, interprétées par le compilateur

Boucle parallèle

```
int i, a[N];
#pragma omp parallel for
for (i = 0; i < N; i++)
    a[i] = 2 * i;
```

OpenMP avantages/inconvénients

+ simple
+ amélioration progressive du code
+ une seule version séquentielle / parallèle
+ peu de modifications sur le code séquentiel d'origine
– exclusivement multiprocesseurs à mémoire partagée
– compilateur dédié
– peu de primitives de synchronisation (atomicité uniquement)
– gros travail si code mal conçu
– introduction de bugs en parallélisant du code non parallélisable

Intel Threading Building Blocks

- Bibliothèque pour C++
- Structures de contrôle optimisées `parallel_for`...
- Structures de données optimisées `concurrent_queue`...
- Peu de primitives de synchronisation (exclusion mutuelle, verrou lecteurs/rédacteurs)
- Implantation spécialisée par modèle de processeur
- Partage de tâches par ≪ vol de travail ≫
- Inconvénient : portabilité limitée (compilateur + matériel)

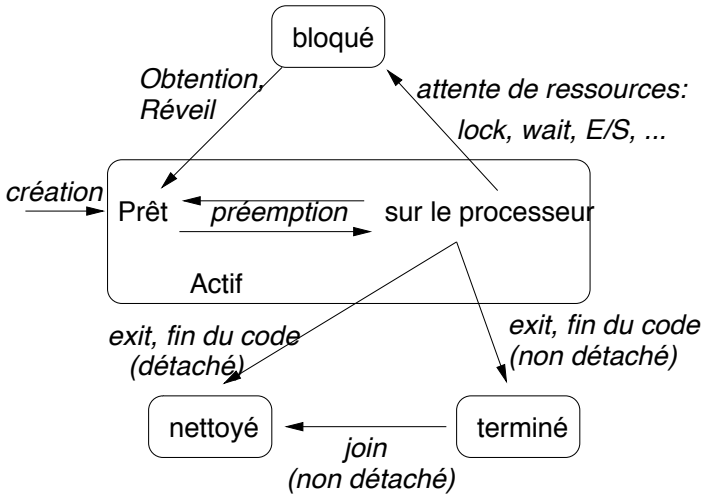
Message Passing Interface (MPI)

- Originellement, pour le calcul haute performance sur clusters de supercalculateurs
- D'un point de vue synchronisation, assimilable aux processus communicants

Plan

1. Retour sur les processus
2. Threads Java
   - Création d'une activité
   - Quelques méthodes
   - Interruption
   - Variables localisées
3. Synchronisation Java
   - Moniteurs
   - Objets de synchronisation
   - Services de régulation du parallélisme
   - Synchronisation de bas niveau/élémentaire
4. Autres environnements
5. Annexe : Threads POSIX

Cycle de vie d'une activité



Création d'une activité

```
int pthread_create (pthread_t *thread,
                    const pthread_attr_t *attr,
                    void * (*start_routine)(void *),
                    void *arg);
```

Crée une nouvelle activité pour exécuter la routine indiquée, appelée avec l'argument `arg`. Les attributs sont utilisés pour définir la priorité et la politique d'ordonnancement (scheduling policy). `thread` contient l'identificateur de l'activité créée.

```
pthread_t pthread_self (void);
int pthread_equal (pthread_t thr1, pthread_t thr2);
```

`self` renvoie l'identificateur de l'activité appelante.
`pthread_equal` : vrai si les arguments désignent la même activité.

Terminaison

```
void pthread_exit (void *status);
```

Termine l'activité appelante en fournissant un code de retour. `pthread_exit(NULL)` est automatiquement exécuté en cas de terminaison du code de l'activité sans appel de `pthread_exit`.

```
int pthread_join (pthread_t thr, void **status);
```

Attend la terminaison de l'activité et récupère le code retour. L'activité ne doit pas être détachée ou avoir déjà été « jointe ».

Terminaison – 2

```
int pthread_detach (pthread_t thr);
```

Détache l'activité `thr`.
Les ressources allouées pour l'exécution d'une activité (pile...) ne sont libérées que lorsque l'activité s'est terminée et que :

- *ou* `join` a été effectué,
- *ou* l'activité a été détachée.

L'activité initiale

Au démarrage, une activité est automatiquement créée pour exécuter la procédure `main`. Elle exécute une procédure de démarrage qui contient le code :

```
{ int r = main(argc,argv); exit(r); }
```

Si la procédure `main` se termine, le processus Unix est ensuite terminé (par l'appel à `exit`), et non pas seulement l'activité initiale. Pour éviter que la procédure `main` ne se termine alors qu'il reste des activités :

- bloquer l'activité initiale sur l'attente de la terminaison d'une ou plusieurs autres activités (`pthread_join`) ;
- terminer explicitement l'activité initiale avec `pthread_exit`, ce qui court-circuite l'appel de `exit`.

Synchronisation

Principe

Moniteur de Hoare élémentaire avec priorité au signaleur :

- verrous
- variables condition
- pas de transfert du verrou à l'activité signalée

Verrou

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutex_attr *attr);

int pthread_mutex_destroy (pthread_mutex_t *m);
```

Verrouillage/déverrouillage

```
int pthread_mutex_lock (pthread_mutex_t *m);
int pthread_mutex_trylock (pthread_mutex_t *m);
int pthread_mutex_unlock (pthread_mutex_t *m);
```

lock verrouille le verrou, avec blocage en attente si déjà verrouillé. Renvoie 0 si ok.

trylock verrouille le verrou si possible et renvoie 0, sinon renvoie `EBUSY` si le verrou est déjà verrouillé.

unlock déverrouille. Seule l'activité qui a verrouillé `m` a le droit de le déverrouiller.

Variable condition

```
pthread_cond_t vc = PTHREAD_COND_INITIALIZER;

int pthread_cond_init (pthread_cond_t *vc,
                       const pthread_cond_attr *attr);

int pthread_cond_destroy (pthread_cond_t *vc);
```

Attente/signal

```
int pthread_cond_wait (pthread_cond_t*,
                       pthread_mutex_t*);
int pthread_cond_timedwait (pthread_cond_t*,
                    pthread_mutex_t*,
                    const struct timespec *abstime);
```

cond_wait l'activité appelante doit posséder le verrou spécifié. L'activité se bloque sur la variable condition après avoir libéré le verrou. L'activité reste bloquée jusqu'à ce que `vc` soit signalée et que l'activité ait réacquis le verrou.

cond_timedwait comme `cond_wait` avec délai de garde. À l'expiration du délai de garde, le verrou est réobtenu et la procédure renvoie `ETIMEDOUT`.

Attente/signal

```
int pthread_cond_signal (pthread_cond_t *vc);
int pthread_cond_broadcast (pthread_cond_t *vc);
```

cond_signal signale la variable condition : une activité bloquée sur la variable condition est réveillée et tente de réacquérir le verrou de son appel de `cond_wait`. Elle sera effectivement débloquée quand elle le réacquerra.

cond_broadcast toutes les activités en attente sont réveillées, et tentent d'obtenir le verrou correspondant à leur appel de `cond_wait`.

Ordonnancement

Par défaut : ordonnancement arbitraire pour l'acquisition d'un verrou ou le réveil sur une variable condition.

Les activités peuvent avoir des priorités, et les verrous et variables conditions peuvent être créés avec respect des priorités.

Données spécifiques

Données spécifiques

Pour une clef donnée (partagée), chaque activité possède sa propre valeur associée à cette clef.

```
int pthread_key_create (pthread_key_t *clef,
                        void (*destructeur)(void *));

int pthread_setspecific (pthread_key_t clef,
                         void *val);
void *pthread_getspecific (pthread_key_t clef);
```

Septième partie

Processus communicants

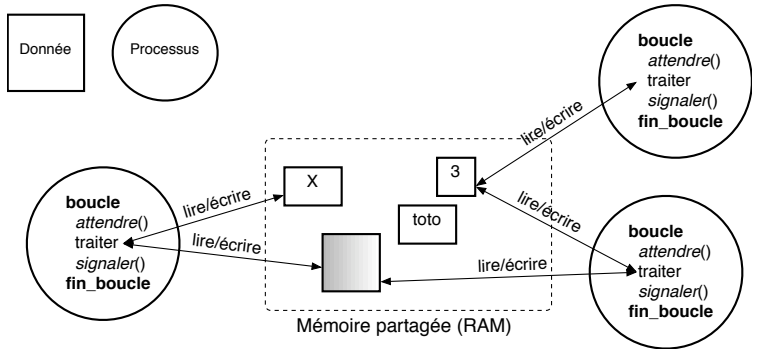
Contenu de cette partie

- Panorama des modèles de programmation concurrente
- Présentation et caractéristiques du modèle des processus communicants
- Outils Ada pour la programmation concurrente
  - Le modèle des processus communicants en Ada : tâches et rendez vous
  - Démarche de conception d'applications concurrentes en Ada
    - Transposition de la démarche vue dans le cadre de la mémoire partagée (moniteurs)
    - Extension tirant parti des possibilités de contrôle fin offertes par Ada

Plan

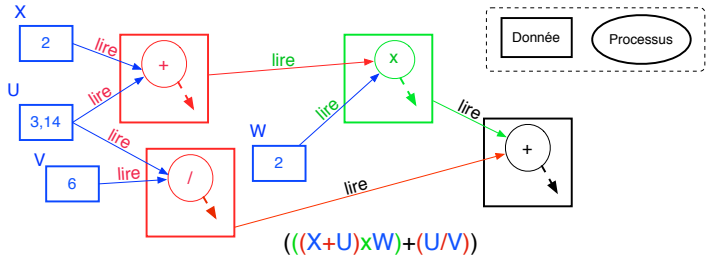
1. Modèles d'interaction pour les programmes parallèles
2. Processus communicants
   - Primitives de communication
   - Synchronisation des processus communicants
3. Ada – Principes
   - Modèle Ada
   - Déclaration d'une tâche
   - Interaction client/serveur : rendez-vous Ada
4. Ada – Méthodologies
   - Construction d'une tâche serveur
   - Exemples
   - Spécification de l'objet partagé par un automate

Modèles d'interaction : mémoire partagée



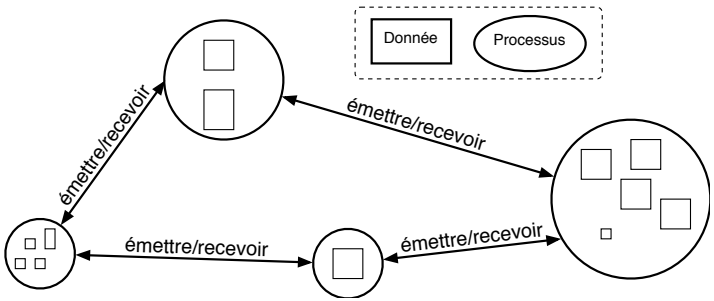
- Communication implicite
  - résulte de l'accès et de la manipulation des variables partagées
  - l'identité des processus n'intervient pas dans l'interaction
- Synchronisation explicite (et nécessaire)
- Architectures/modèles cibles
  - multiprocesseurs à mémoire partagée,
  - programmes multiactivités

Modèles d'interaction : données actives



- Chaque donnée est évaluée par un processus dédié
- Communication = lecture
- Synchronisation = attente de disponibilité des données
- Architectures/modèles cibles
  - parallélisme massif, à grain fin :
    - programmation par flots de données (streams Java 8),
    - programmation fonctionnelle (Scala)
  - objets immuables

Modèles d'interaction : processus communicants



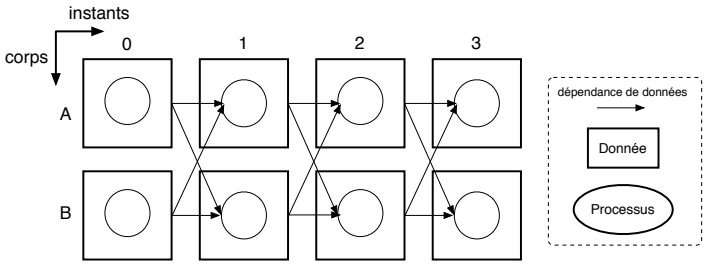
- Données encapsulées par les processus
- Communication nécessaire, explicite : échange de messages
  - Programmation et interactions plus lourdes
  - Visibilité des interactions → possibilité de trace/supervision
- Synchronisation implicite : attente de disponibilité des messages
- Les processus déterminent la granularité du parallélisme
- Architectures/modèles cibles
  - systèmes répartis : sites distants, reliés par un réseau
  - moniteurs, tâches Ada, API messages : sockets, MPI, JMS...

Exemple : simulation discrète de corps en interaction

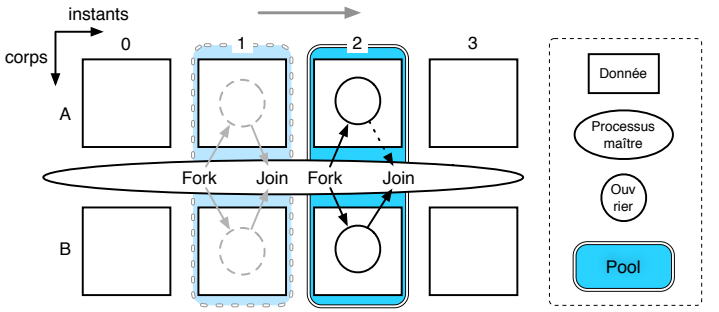
Problème des N corps

Calculer la position de N corps C_1, $C_2 \ldots C_N$ à une suite d'instants successifs t_0, t_1, $\ldots t_k$. La position d'un corps à l'instant t_i est déterminée par la position de l'ensemble des corps à l'instant t_{i-1}.

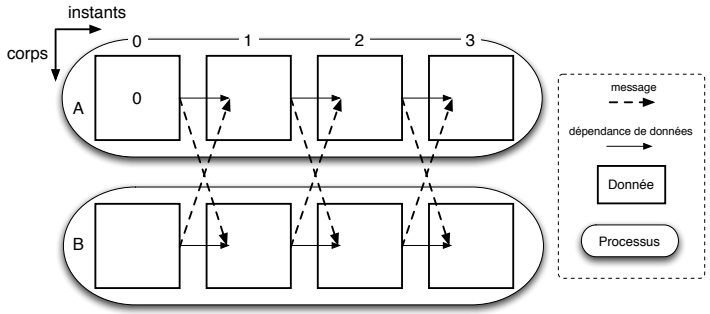
- **Parallélisme/résultat** (Flots de données/Structures actives)
  - matrice corps x instants
  - un processus d'évaluation par élément de la matrice



- **Parallélisme planifié** (Structures partagées + pool d'ouvriers)
  - données partagées : positions aux instants précédents
  - un maître coordonne les ouvriers (schéma fork/join)
  - N ouvriers : chaque ouvrier évalue l'élément de la colonne courante qui lui est affecté
  - progression colonne par colonne : le maître attend qu'une colonne soit complètement évaluée (`join`), avant de lancer l'évaluation de la suivante (`fork`)
  - peut être vu comme une réalisation du parallélisme résultat



- **Parallélisme/spécialistes** (Messages/Clients-Serveur)
  - un processus par corps
  - accès aux positions passées des autres corps → messages



Plan

1 Modèles d'interaction pour les programmes parallèles

2 Processus communicants
- Primitives de communication
- Synchronisation des processus communicants

3 Ada – Principes
- Modèle Ada
- Déclaration d'une tâche
- Interaction client/serveur : rendez-vous Ada

4 Ada – Méthodologies
- Construction d'une tâche serveur
- Exemples
- Spécification de l'objet partagé par un automate

Processus communicants

Synchronisation obtenue via des primitives de communication bloquantes : envoi (bloquant) de messages / réception bloquante de messages

- Communicating Sequential Processes (CSP) / Calculus of Communicating Systems (CCS) / π-calcul
- Ada

Les principes détaillés des échanges et leur utilisation pour développer des applications sont vus dans l'UE ≪ intergiciels ≫.
On intéresse ici avant tout à la synchronisation.

Opérations

- Emettre(message, destination)
- Recevoir(message, source)

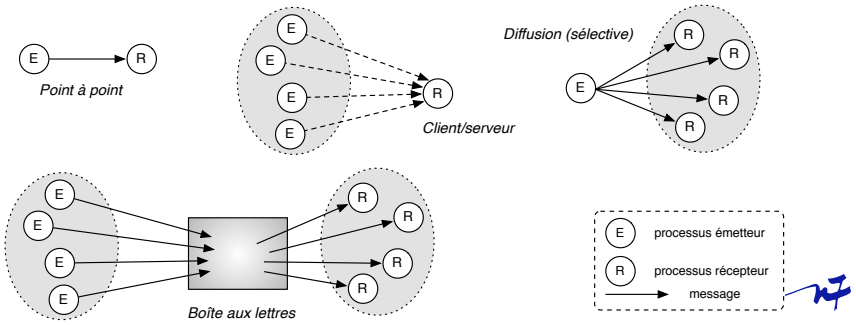
Synchronisation liée aux opérations *du modèle des proc. communicants*

- Réception bloquante : attente d'un message
- L'émission peut être, selon les modèles
  - non bloquante (comm. asynchrone) : l'émission termine dès la prise en charge du message par le medium de communication.
  - bloquante (communication synchrone) : l'émetteur doit attendre jusqu'à la réception effective du message
    → rendez-vous entre l'émetteur et le destinataire

Désignation des activités sources et destinataires

La source ou le destinataire peuvent être une ou plusieurs activités → différents schémas de communication :

- communication directe, « point à point » (1-1) entre activités
- diffusion (sélective) (1-n)
- communication indirecte : via une « boite aux lettres » (un « canal ») attachée à un processus (n-1 : port), ou partagée (n-m)



Exemple : le langage CSP (Communicating Sequential Processes)

Caractéristiques

- émission bloquante
- échanges de messages via des canaux explicitement désignés, attachés ou non à un processus.

Opérations

- envoi d'un message `msg` sur le canal `c` : $c!msg$
- réception d'un message `msg` sur `c` : $c?msg$

Rendez-vous

L'émission et la réception sont bloquantes : chaque communication est un rendez-vous entre un proc. émetteur et un proc. récepteur :
$proc\ p\ \{c!m\}\ \|\ proc\ q\ \{c?m\}$

Alternative

Définit un choix à effectuer parmi un ensemble de communications possibles :

- choix entre deux réceptions : $c_1?m_1[]c_2?m_2$
- choix entre deux émissions : $c_1!m_1[]c_2!m_2$
- une alternative peut comporter plus de deux choix (sic)
- une alternative peut comporter des émissions et des réceptions
- les choix peuvent être gardés par une condition g : $g \rightarrow s?m$

Evaluation

Les choix dont les gardes sont fausses sont éliminés. Ensuite :

- Aucun des choix restants n'est possible immédiatement → attendre que l'un d'eux le devienne
- Un seul choix possible → le faire
- Plusieurs choix possibles → sélection non-déterministe (arbitraire)

Synchronisation des processus communicants

- La démarche de résolution des problèmes de synchronisation vue pour le modèle de la mémoire partagée est basée sur la définition et le contrôle de l'interaction avec un objet partagé.
- Le modèle des processus communicants fournit une base à l'encapsulation des données/ressources :

  le seul moyen d'accéder aux données locales à un processus est d'échanger des messages avec ce processus

→ La démarche vue pour le modèle de la mémoire partagée se transpose simplement dans le contexte des processus communicants :
Définir un processus arbitre (ou « serveur »), encapsulant l'objet partagé, pour contrôler et réaliser les opérations sur celui-ci.

Mise en œuvre d'un processus arbitre pour un objet partagé

Interactions avec l'objet partagé (protocole requête/réponse) :
Pour chaque opération $\mathcal{O}p$,

- émettre un message de requête vers l'arbitre
- attendre le message de réponse de l'arbitre
  ($\Rightarrow$ se synchroniser avec l'arbitre)

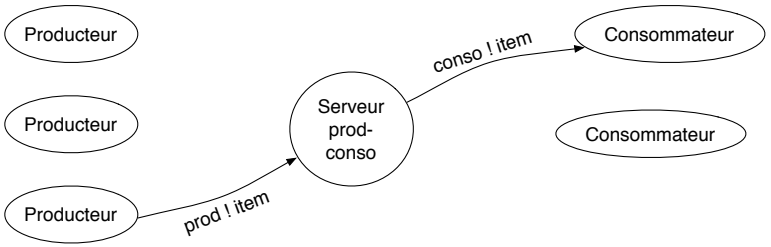
$\rightarrow$ en CSP :

- messages échangés via un canal $\mathcal{C}\_\mathcal{O}p$ associé à l'opération $\mathcal{O}p$,
- interaction = rendez-vous entre le client et l'arbitre, sur $\mathcal{C}\_\mathcal{O}p$

Schéma de fonctionnement de l'arbitre

- L'arbitre exécute une boucle infinie contenant une alternative
- Cette alternative possède une branche par opération fournie
- Chaque branche est gardée par la condition d'acceptation de l'opération (suivie de l'attente du rendez-vous correspondant)

Exemple 1 (CSP) : producteurs/consommateurs (1/3)



Objet partagé :

Tampon borné de taille N

Opérations sur le tampon partagé :

- Déposer $\rightarrow$ canal associé : *prod* (sens producteur $\rightarrow$ serveur)
- Retirer $\rightarrow$ canal associé : *conso* (sens serveur $\rightarrow$ consommateur)

Producteurs/consommateurs (2/3)

Producteur

```
Processus producteur
  boucle
    ...
    item := ...
    prod!item
    ...
  finboucle
```

Consommateur

```
Processus consommateur
  boucle
    ...
    conso?item
    utiliser item
    ...
  finboucle
```

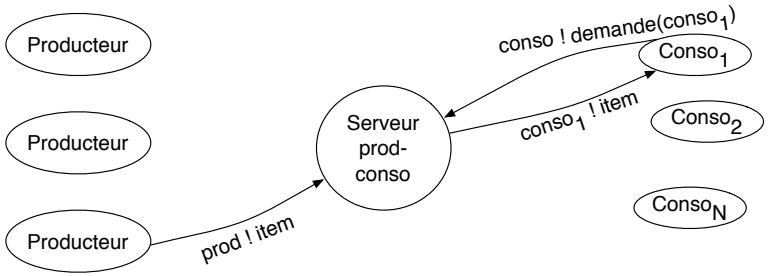
Producteurs/consommateurs (3/3)

Activité de synchronisation (serveur)

```
Processus prod-conso
  var
    int nbocc := 0;
    Item m;
    File<Item> tampon;
  boucle
     nbocc < N → prod?m; nbocc++; tampon.ranger(m);
  □
     nbocc > 0 → ; conso!tampon.extraire(); nbocc--;
  finboucle
```

- Nécessite l'alternative mixte
  (plus complexe à réaliser que l'alternative en réception)
- Nécessite la réception multiple
  (plusieurs activités partagent le même canal *conso*)

[Producteurs/consommateurs : variante (1/3)]



- Un canal *prod* pour les demandes de dépôt
- Un canal *conso* pour les demandes de retrait
- Pour chaque activité $Conso_i$ demandant un retrait, un canal $conso_i$, pour la réponse à la demande de retrait

[Producteurs/consommateurs : variante (2/3)]

Producteur

```
Processus producteur
  boucle
    ...
    item := ...
    prod!item
    ...
  finboucle
```

Consommateur

```
Processus consommateur
Canal moi
  boucle
    ...
    conso!demande(moi);
    moi?item;
    utiliser item
  finboucle
```

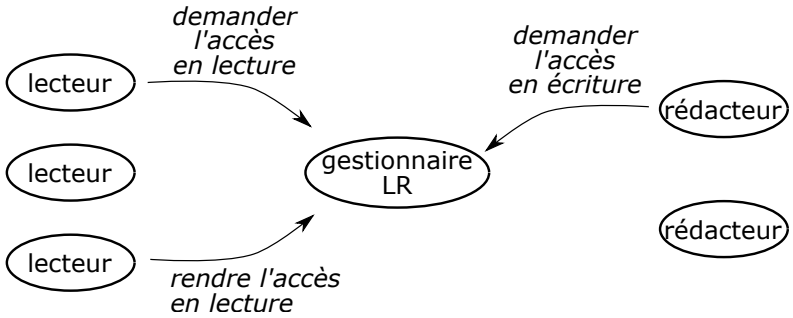

[Producteurs/consommateurs : variante (3/3)]

Processus de synchronisation

```
Processus prod-conso
  var
    int nbocc := 0;
    Item m;
    File<Item> tampon;
    Canal c;
  boucle
     nbocc < N → prod?m; nbocc++; tampon.ranger(m)
  □
     nbocc > 0 → conso?demande(c); m := tampon.extraire();
                 c!m; nbocc--;
  finboucle
```

Nécessite des variables *Canal*, pouvant être passées en paramètre.

Exemple 2 (CSP) : lecteurs/rédacteurs (1/3)



- Un canal pour chaque opération : *DE*, *TE*, *DL*, *TL*
- Émission bloquante ⇒ contrôle des requêtes par l'arbitre : un message n'est accepté que si l'état de l'objet partagé l'autorise

Lecteurs/rédacteurs (2/3)

Utilisateur

```
Processus utilisateur
  boucle
    DL!_;  // demander lecture
    ...
    TL!_;  // terminer lecture
    ...
    DE!_;  // demander écriture
    ...
    TE!_;  // terminer écriture
  finboucle
```

Lecteurs/rédacteurs (3/3)

Processus de synchronisation

```
Processus SynchroLR
  var
    int nblec = 0;
    boolean ecr = false;
  boucle
     nblec = 0 ∧ ¬ecr → DE?_; ecr := true;
  □
     ¬ecr → DL?_; nblec++;
  □
     TE?_; ecr := false;
  □
     TL?_; nblec--;
  finboucle
```

Plan

1. Modèles d'interaction pour les programmes parallèles
2. Processus communicants
   - Primitives de communication
   - Synchronisation des processus communicants
3. Ada – Principes
   - Modèle Ada
   - Déclaration d'une tâche
   - Interaction client/serveur : rendez-vous Ada
4. Ada – Méthodologies
   - Construction d'une tâche serveur
   - Exemples
   - Spécification de l'objet partagé par un automate

Modèle Ada

Modèle orienté vers une organisation en termes de processus communicants

- Application (processus lourd) = ensemble de *tâches* (activités) concurrentes
- Interactions entre tâches privilégiant le schéma client-serveur.
- Contrôle fin de l'ordonnancement des requêtes au niveau du serveur.
- Transposable simplement à un environnement réparti.

Déclaration d'une tâche

Schéma d'interaction privilégié : client-serveur
→ toute tâche exporte une interface (*points d'entrée*)

Exemple

```
task ProdCons is
   entry Deposer (msg: in T);
   entry Retirer (msg: out T);
end ProdCons;
```

Syntaxe

```
task [type] <nom> is
   { entry <point d'entrée> (<param formels>); }+
end <nom> ;
```


Modularité → interface et implémentation (*corps*) déclarées séparément

Exemple

```
task body ProdCons is
   Libre : integer := N;
begin
     ⋮
     accept Deposer (msg : in T) do
          deposer_dans_tampon(msg);
     end Deposer;
     Libre := Libre - 1;
     ⋮
end ProdCons;
```

Syntaxe

```
task body <nom> is
  [ < déclaration des variables locales > ]
begin
   [ < instructions > ]
end < nom > ;
```

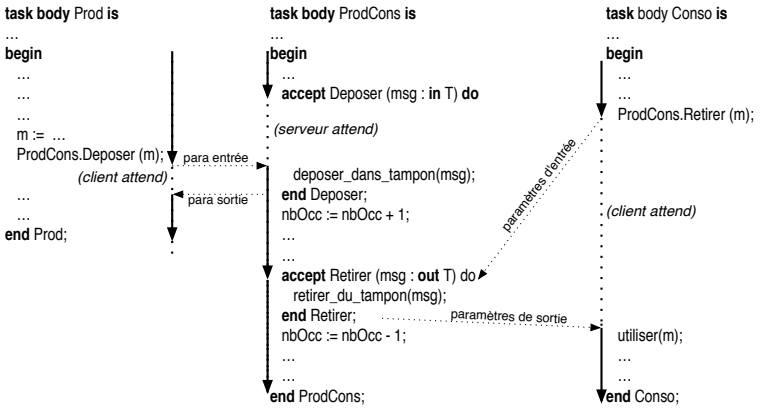

Création (activation) des tâches

Une tâche peut être activée :

- statiquement : une `task T` déclarée directement, ou comme instance d'un type de tâche, est créée au démarrage du programme, avant l'initialisation des tâches qui utilisent `T.entry`.
- dynamiquement :
  - déclaration par `task type T`
  - activation par allocation : `var t is access T := new T;`
  - possibilité d'activer plusieurs tâches d'interface `T`.

Interaction client/serveur : rendez-vous Ada

- tâche cliente : requête de service = appel d'un point d'entrée
  Exemple : `tampon.déposer(ln)`
- tâche serveur : prise en charge d'une requête → instruction accept



Protocole

- Si un client demande un rendez-vous alors que le serveur n'est pas prêt à l'accepter, le client se bloque en attente de l'acceptation.
- Si un serveur est prêt à accepter un rendez-vous et qu'il n'y a pas de demandeur, il se bloque.
- L'appelant est bloqué pendant l'exécution du corps du rendez-vous.

Côté client : appel d'entrée (≈ demande de rendez-vous)

```
<nom tâche>.<nom d'entrée> (<param effectifs>);
```

Similaire à un appel de procédure.

Exemple

```
tampon.déposer(ln)
```

Acceptation (côté serveur)

```
accept <nom d'entrée> (<param formels>)
     [ do
         { <instructions> }+
       end <nom d'entrée> ]
```

Exemple

```
accept Deposer (msg : in T) do
     deposer_dans_tampon(msg);
     Libre := Libre - 1;
end Deposer;
```

Remarques

- les `accept` ne peuvent figurer que dans le corps des tâches
- `accept` sans corps → synchronisation « pure »
- une file d'attente (FIFO) est associée à chaque entrée
  l'attribut `'COUNT` donne pour chaque entrée, la taille de la file
- la gestion et la prise en compte des appels diffèrent par rapport aux moniteurs
  - la prise en compte d'un appel au service est déterminée par le serveur → serveur « actif »
  - plusieurs appels à un même service peuvent déclencher des traitements différents
  - le serveur peut être bloqué, tandis que des clients attendent
- échanges de données :
  - lors du début du rendez-vous, de l'appelant vers l'appelé ;
  - lors de la fin du rendez-vous, de l'appelé vers l'appelant.

Plan

1. Modèles d'interaction pour les programmes parallèles
2. Processus communicants
   - Primitives de communication
   - Synchronisation des processus communicants
3. Ada – Principes
   - Modèle Ada
   - Déclaration d'une tâche
   - Interaction client/serveur : rendez-vous Ada
4. Ada – Méthodologies
   - Construction d'une tâche serveur
   - Exemples
   - Spécification de l'objet partagé par un automate

Construction d'une tâche serveur

Alternative gardée (`select`)

Ada permet à un serveur d'attendre un appel à un point d'entrée quelconque parmi un ensemble donné de points d'entrée.

```
select
    when Libre > 0 =>
       accept Deposer (msg : in T) do
          deposer_dans_tampon(msg);
       end Deposer;
        ...
  or
    when Libre < N =>
       accept Retirer (msg : out T) do
          msg := retirer_du_tampon();
       end Retirer;
        ...
  or ...
end select;
```

Evaluation du `select`

- évaluer les conditions vraies → branches ouvertes
- branche ouverte et appel en attente → branche franchissable
- si des branches sont franchissables, choisir (arbitrairement) une branche parmi celles-ci
- sinon attendre qu'une branche ouverte soit franchissable

Compléments

- `when` omis = true
- clauses/branches particulières
  - clause `else` possible, comme dernière possibilité d'un `select`
    - exécutée si aucune branche n'est franchissable
    - évite le blocage en réception
    - clause `else` omise et aucune branche ouverte → exception `program_error`
  - `when <condition> => delay <nbDeSecondes>` exécutée si ouverte et aucune branche n'est franchissable à l'issue du délai
  - `when <condition> => terminate` termine le serveur en l'absence de tâches clientes potentielles.

Structure d'un serveur

Serveur ≪ type ≫

`select` dans une boucle sans fin
→ tâche (passive) dédiée à la réalisation de services

Remarque

Similitude entre cette structure de tâche et les moniteurs

→ possibilité de réutiliser/transposer directement directement la démarche de conception et les techniques d'optimisation vues pour les moniteurs.

Terminaison du serveur

Une tâche T est potentiellement appelante de T' si

- T' est une tâche statique et le code de T contient au moins une référence à T',
- ou T' est une tâche dynamique et (au moins) une variable du code de T référence T'.

Une tâche se termine quand :

- elle atteint la fin de son code,
- ou elle est bloquée en attente de rendez-vous sur un `select` avec clause `terminate` et toutes les tâches potentiellement appelantes sont terminées.

Exemple 1 : producteurs/consommateurs

Client : utilisation

```
begin
   -- engendrer le message m1
   ProdCons.Deposer (m1);
   -- ...
   ProdCons.Retirer (m2);
   -- utiliser m2
end
```


```
task body ProdCons is
   Libre : integer := N;
begin
   loop
      select
         when Libre > 0 =>
            accept Deposer (msg : in T) do
               deposer_dans_tampon(msg);
            end Deposer;
            Libre := Libre - 1;
      or
         when Libre < N =>
            accept Retirer (msg : out T) do
               msg := retirer_du_tampon();
            end Retirer;
            Libre := Libre + 1;
      or
         terminate;
      end select;
   end loop;
end ProdCons;
```

Exemple 2 : allocateur de ressources multiples

Un système comporte des ressources critiques c'est-à-dire non partageables et non préemptibles, comme des pages mémoire. Le service d'allocation de ressources multiples permet à un processus d'acquérir par une seule action plusieurs ressources. L'exemple ne traite que la synchronisation et ne présente pas la gestion effective des (identifiants de) ressources.

Déclaration du serveur

```
task Allocateur is
   entry demander (nbDemandé: in natural;
                   id : out array of RessourceId);
   entry rendre (nbRendu: in natural;
                 id : in array of RessourceId);
end Allocateur;
```


```
task body Allocateur is
   nbDispo : integer := N;
begin
   loop
      select
          accept Demander (nbDemandé : in natural) do
              while nbDemandé > nbDispo loop
                  accept Rendre (nbRendu : in natural) do
                      nbDispo := nbDispo + nbRendu;
                  end Rendre;
              end loop;
              nbDispo := nbDispo - nbDemandé;
          end Demander;
      or
          accept Rendre (nbRendu : in natural) do
              nbDispo := nbDispo + nbRendu;
          end Rendre;
      or
          terminate;
      end select;
   end loop;
end Allocateur;
```

Extension de la méthodologie : définition d'un automate

Idée

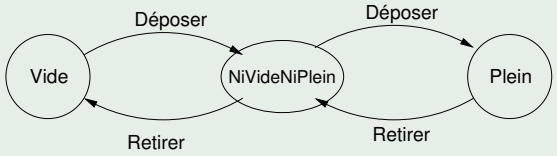
Affiner la définition des utilisations cohérentes d'un objet partagé par un ensemble de processus concurrents.

- Démarche moniteurs basée sur la définition de l'ensemble des états possibles (invariant du moniteur)
- Compléter cette définition par une fonction de transition, précisant, à partir de chaque état possible, quelles sont les actions (transitions) possibles, et quel est l'état résultat
  - → définition d'un automate
  - → le serveur traite les requêtes conformément à cet automate

Démarche de construction de l'automate

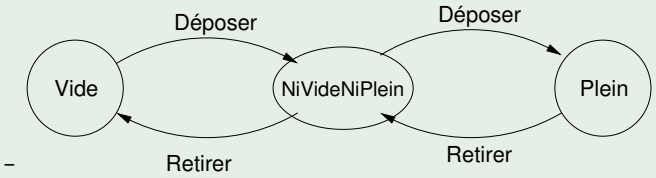
- identifier les états de l'objet partagé géré par le serveur
- pour chaque état, identifier les rendez-vous acceptables
- un rendez-vous accepté change (éventuellement) l'état

Producteurs/Consommateurs à 2 cases




```
task body ProdCons is
   type EtatTampon is (Vide, NiVideNiPlein, Plein);
   etat : EtatTampon := Vide;
begin
   loop
     case etat is
     when NiVideNiPlein =>
            select
               accept Deposer (msg : in T) do
                  deposer_dans_tampon(msg);
               end Deposer;
               etat := Plein;
            or
               accept Retirer (msg : out T) do
                  msg := retirer_du_tampon();
               end Retirer;
               etat := Vide;
            end select;
```

Producteurs/Consommateurs à 2 cases (suite)

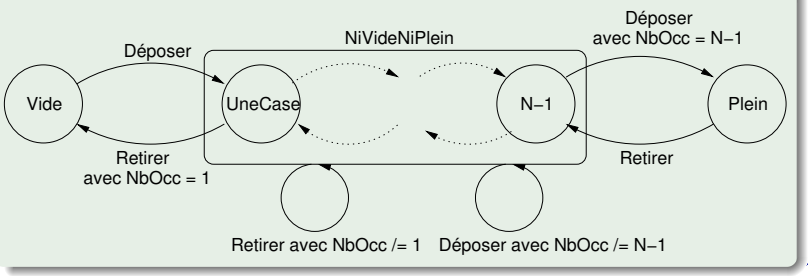



```
   --
     when Vide =>
            accept Deposer (msg : in T) do
               deposer_dans_tampon(msg);
            end Deposer;
            etat := NiVideNiPlein;
     when Plein =>
            accept Retirer (msg : out T) do
               msg := retirer_du_tampon();
            end Retirer;
            etat := NiVideNiPlein;
     end case;
   end loop;
end ProdCons;
```


Automate paramétré

Un ensemble d'états peut être représenté comme un état *paramétré*
Les valeurs du paramètre différenciant les états de l'ensemble peuvent être utilisées pour étiqueter les transitions, lorsqu'elles diffèrent selon l'état de l'ensemble.

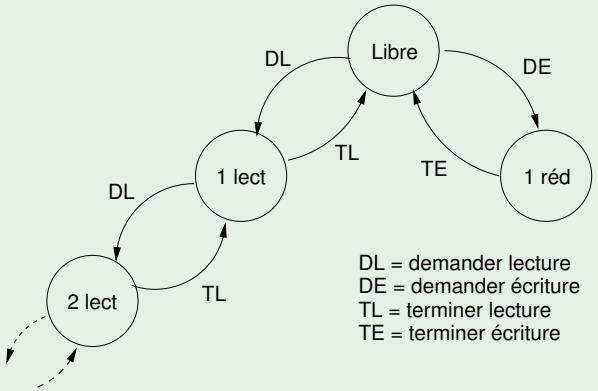
Producteurs/Consommateurs à N cases



Exercice

Réaliser un serveur (basé sur un automate) gérant les accès à un fichier partagé selon le schéma lecteurs rédacteurs, avec priorité aux lecteurs.

Indication : automate développé pour le schéma L/R, sans priorités



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 (possibilité d'annuler les effets d'un traitement)
  - Isolation (non interférence entre traitements)
- Adaptation de la notion de transaction au modèle de la programmation concurrente avec mémoire partagée (mémoire transactionnelle)

Plan

1 Transactions
- Concurrence et cohérence
- Service transactionnel

2 Atomicité

3 Isolation : contrôle de concurrence
- Principe
- Modélisation
- Résultat fondamental
- Méthodes de contrôle de concurrence
- Méthodes optimistes : certification
- Méthodes pessimistes

4 Mémoire transactionnelle
- Motivation
- Intégration aux langages de programmation
- Réalisation
- Questions ouvertes

Comment garantir la cohérence d'activités concurrentes ?

Situation

≪ objet ≫ partagé, utilisé simultanément par plusieurs processus

Problème

garantir que cet objet est ≪ correctement utilisé ≫

Points de vues/approches possibles

- mise en œuvre directe : *synchronisation*, coordination des actions des différents processus, contrôlant explicitement l'attente/la progression des processus
- utilisation d'un service/abstraction général : *concurrence*. Dans ce cas, chaque processus interagit avec l'objet partagé comme s'il était seul à l'utiliser : le partage est transparent.

Cohérence et concurrence dans le contexte des SGBD

Contexte : traitements concurrents / données partagées

- données partagées, existant indépendamment des traitements
- système ouvert : les traitements ne sont pas connus a priori
  → chaque traitement doit pouvoir être conçu indépendamment

→ approche analogue à celle suivie pour la synchronisation :

- Caractérisation des utilisations concurrentes correctes (cohérentes) par un ensemble d'états possibles/permis pour les données partagées, que tout traitement doit respecter.
- Cet ensemble est défini en intention par un prédicat d'état : contrainte(s) d'intégrité, invariant

Abstraction de la concurrence : service transactionnel

Service de gestion des accès concurrents aux données partagées

- basé sur la notion d'état cohérent
- déclaratif : le programmeur doit simplement indiquer les traitements (transactions) pour lesquels la cohérence doit être garantie par le service transactionnel.

Définition de base : transaction

Suite d'opérations qui, exécutée seule
à partir d'un état initial cohérent, aboutit à un état final cohérent

Domaines d'utilisation

- Systèmes d'information : bases de données → intergiciels
- Mémoire transactionnelle (architectures mutiprocesseurs) (HTM/STM = hardware/software transactional memory)

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)

Contrat du service transactionnel : propriétés ACID

Cohérence	toute transaction maintient les contraintes d'intégrité La validité sémantique est du ressort du programmeur
Isolation	pas d'interférences entre transactions : les états intermédiaires d'une transaction ne sont pas observables par les autres transactions. → modularité
Atomicité	ou « tout ou rien » : en cas d'abandon (volontaire ou subi) tous les effets d'une transaction sont annulés
Durabilité	permanence des effets d'une transaction validée

Exemple : base de données bancaires

- Données partagées : ensemble des comptes (X,Y ...)
- Contraintes :
  - la somme des comptes est constante (X.val + Y.val = Cte)
  - chaque compte a un solde positif (X.val ≥ 0 et Y.val ≥ 0)
- Transaction :
  virement d'une somme S du compte X au compte Y

```
tdébut
  si S > X.val alors
    "erreur" ;
  sinon
    X.val := X.val - S;
    Y.val := Y.val + S;
  finsi ;
tfin
```

Remarques :

- les états intermédiaires ne sont pas forcément cohérents
- expression déclarative : parenthésage par `tdébut/tfin`

Plan

1. Transactions
   - Concurrence et cohérence
   - Service transactionnel
2. Atomicité
3. Isolation : contrôle de concurrence
   - Principe
   - Modélisation
   - Résultat fondamental
   - Méthodes de contrôle de concurrence
   - Méthodes optimistes : certification
   - Méthodes pessimistes
4. Mémoire transactionnelle
   - Motivation
   - Intégration aux langages de programmation
   - Réalisation
   - Questions ouvertes

Atomicité (tout ou rien)

Objectif

- Intégrer les résultats des transactions « bien » terminées
- 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.

Mise en œuvre de l'atomicité

Opérations de base

- *défaire* : revenir à l'état initial d'une transaction annulée
- *refaire* : restaurer l'état atteint par une transaction annulée temporairement ou une (in)validation interrompue

Réalisation des opérations *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* → utiliser les avant pour revenir à l'état initial
  - *refaire* → 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 optimiste : propagation en continu

Utilisation d'un journal des valeurs avant

- técrire → écriture directe en mémoire permanente
- valider (tfin) → effacer les images avant
- défaire (tabandon) → 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))
(3) tabandon()	(4) tabandon() → x=10 au lieu de x=5

Remède (?) : bloquer les accès en conflit avec técrire(x,-) → pas de parallélisme

Approche pessimiste : propagation différée

Utilisation d'un journal des valeurs après

Principe

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

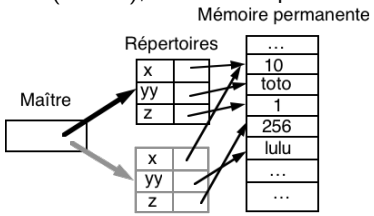
- écrire → préécriture, dans l'espace de travail
- valider → recopier l'espace de travail en mémoire stable (*liste d'intentions*), puis copier celle-ci en mémoire permanente
  → protection contre les pannes en cours de validation
- défaire → libérer l'espace de travail
- refaire → reprendre la recopie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

Technique sans journal (SGBD/SGF) : pages d'ombre

Principe

- Hypothèse : les blocs de données sont accessibles indirectement, via des blocs/pages d'index
- Chaque transaction dispose d'une copie des blocs d'index

- técrire → écriture en mémoire rémanente, via la copie des index
- défaire → purger la copie
- valider → remplacer l'original par la copie, de manière atomique
  Garantir l'atomicité → pages d'ombre : la copie valide est repérée par un enregistrement (maître), écrit atomiquement



- refaire → sans objet (validation atomique)

Plan

1 Transactions
- Concurrence et cohérence
- Service transactionnel

2 Atomicité

3 Isolation : contrôle de concurrence
- Principe
- Modélisation
- Résultat fondamental
- Méthodes de contrôle de concurrence
- Méthodes optimistes : certification
- Méthodes pessimistes

4 Mémoire transactionnelle
- Motivation
- Intégration aux langages de programmation
- Réalisation
- Questions ouvertes

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 (si possible)

→ recherche d'un résultat final identique à celui qui aurait été obtenu en exécutant les transactions en exclusion mutuelle

Terminologie

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

Remarque : Il peut exister plusieurs exécutions sérialisées équivalentes ; il suffit qu'il en existe au moins une.

Comment vérifier la sérialisabilité ?

Problème

On considére une exécution concurrente $(T_1||T_2||\ldots||T_n)$ d'un ensemble de transactions $\{T_1,\ T_2\ \ldots\ T_n\}$.
Cette exécution donne-t-elle le même résultat que l'une des exécutions en série $(T_1\,;\ T_2\,;\ \ldots\ ;\ T_n)$, ou $(T_2\,;\ T_1\,;\ \ldots\ ;\ T_n)$, ... ?

Idée

- le résultat de l'exécution de $(T_1||T_2||\ldots||T_n)$ sera celui d'un entrelacement des opérations de $\{T_1,\ T_2\ \ldots\ T_n\}$.
- si les différents entrelacements donnent le même résultat, alors toutes les exécutions série, et toutes les exécutions de $(T_1||T_2||\ldots||T_n)$ donneront le même résultat.
  $\Rightarrow$ pour vérifier la sérialisabilité, on peut se limiter aux opérations dont l'ordre d'exécution influence le résultat.

Notion de conflit

L'ordre d'exécution change-t-il le résultat ?

- `x := y/2  || u := w + v` → non
- `x := y/2  || y := y + 1` → oui
- `y := y + 1 || y := y + 1` → non

→ opérations en *conflit* :

opérations non commutatives exécutées sur un même objet

Exemple principal : 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') ;

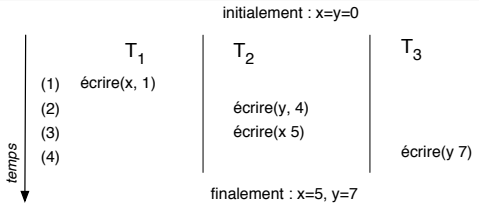
Remarque : la notion de conflit n'est pas spécifique à *lire/écrire*

→ généralisation : définir un tableau de commutativité entre actions
Exemple : lire, écrire, incrémenter, décrémenter

Graphe de dépendance

Idée : Les conflits déterminent l'ordre série équivalent, s'il existe.

Exemple



Dans toute exécution série donnant le même résultat,

- on doit trouver T_1 avant T_2 (sinon, x=1 au final)
- on doit trouver T_2 avant T_3 (sinon, y=4 au final)

Règle générale : s'il existe une exécution série S donnant le même résultat qu'une exécution concurrente $C = (T_1||T_2||\ldots||T_n)$, **alors** lorsqu'une opération op_i de T_i est en conflit avec une opération op_k de T_k, et que op_i a été exécutée avant op_k, T_i se trouve nécessairement avant T_k dans S (sinon le résultat final de S serait différent de celui de C)

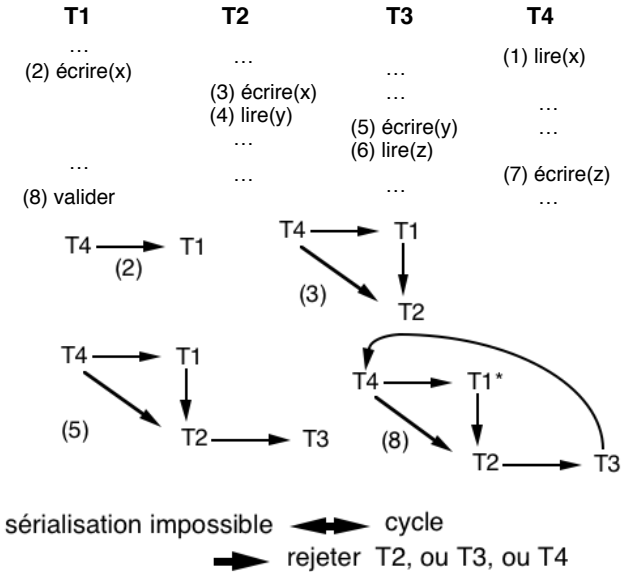
Définitions

- Relation de dépendance → : $T_1 \to T_2$ ssi une opération de T_1 précède et est en conflit avec une opération de T_2.
- Graphe de dépendance : relations de dépendance pour les transactions déjà validées.

Théorème (Critère de sérialisabilité [Papadimitriou])

Exécution sérialisable ⇔ son graphe de dépendance est acyclique.

Exemple



Méthodes de contrôle de concurrence

Quand vérifier la sérialisabilité ?

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

Comment garantir la sérialisabilité ?

- utilisation explicite du graphe de dépendance (coûteux)
- définir un ordre sur les transactions ($\rightarrow$ acyclicité) et bloquer/rejeter toute (trans)action introduisant une dépendance allant à l'encontre de cet ordre
  - ordre arbitraire $\rightarrow$ estampilles
  - ordre chronologique d'accès $\rightarrow$ verrous (méthodes continues)

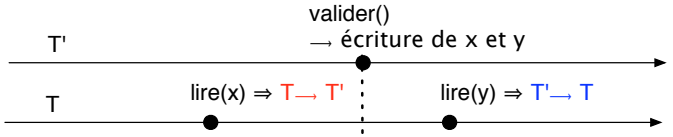
Contrôle de concurrence par certification : principe

- **Hypothèse** : écritures en mémoire privée avec recopie à la validation
- ordre de sérialisation = ordre de validation
- une transaction valide s'il est certain que ses conflits avec les transactions ayant déjà validé suivent l'ordre de validation

T demande à valider $\rightarrow$ écritures de T pas encore effectives

Conflits possibles entre T et les transactions validées :

- tous les conflits EE suivent l'ordre de validation : les écritures validées précèdent celles de T (qui n'ont pas eu lieu)
- conflits LE
  - les lectures validées précèdent toujours les écritures de T
  - mais il n'est pas certain que les écritures validées précèdent toujours les lectures de T



Contrôle de concurrence par certification : algorithme

Algorithme

```
C : nb de transactions certifiées  (ordonne les transactions)
T.déb, T.fin : valeurs de C au début et 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'.ecrits = ∅)
alors
    C ← C + 1
    T.val ← C
sinon
    tabandon(T)
finsi
```

Contrôle continu : estampilles

ordre de sérialisation = ordre des estampilles

→ pour toute transaction T, les accès de T doivent passer après ceux de toutes les transactions d'estampille inférieure à celle de T

Algorithme

```
T.E : estampille de T
O.lect : estampille du plus «récent» (grand) lecteur de O
O.réd : estampille du plus «récent» (grand) écrivain de O

procédure lire(T,O)                 | procédure écrire(T,O,v)
si T.E ≥ O.réd                      | si T.E ≥ O.lect ∧ T.E ≥ O.réd
alors /* lecture de O possible */   | alors /* écriture de O possible */
   lecture effective                |    écriture effective
   O.lect ← max(O.lect,T.E)         |    O.red ← T.E
sinon                               | sinon
   abandon de T                     |    abandon de T
finsi                               | finsi
```

Estampille fixée au départ de la transaction ou au premier conflit.

Estampilles : remarques (1/2)

Amélioration : réduire les cas d'abandons.

Algorithme (règle de Thomas)

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


Estampilles : remarques (2/2)

Les opérations `lire(...)` et `écrire(...)` peuvent devoir être complétées/adaptées, en fonction de la politique de propagation :

- propagation continue (optimiste)
  → gérer les abandons en cascade
- propagation différée (pessimiste)
  → les écritures effectives n'ont lieu qu'en fin de transaction.
  Par conséquent
  - les estampilles d'écriture (`O.red`) ne peuvent être fixées qu'au moment de la validation
  - les tests relatifs aux opérations d'écriture doivent être (ré)effectués à la terminaison de la transaction

Contrôle continu : verrouillage à deux phases

Verrous en lecture/écriture :
si $T_1 \rightarrow T_2$, T_2 est bloquée jusqu'à ce que T_1 valide.

Ordre de sérialisation = ordre chronologique d'accès aux objets

Si toute transaction est

- bien formée (prise du verrou avant tout accès)
- à deux phases (pas de prise de verrou après une libération)
  `phase 1 : acquisitions et accès`
  `< point de verrouillage maximal >`
  `phase 2 : libérations`

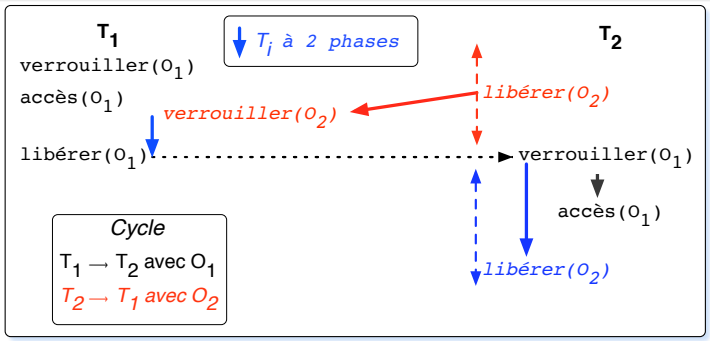
alors la sérialisation est assurée.

Ordre série = ordre d'apparition des points de verrouillage maximaux
(ordre des validations, dans le cas de 2PL strict, cf infra)

Verrouillage à deux phases : justification du protocole

Idée de base

Lorsque 2 transactions sont en conflit, tous les couples d'opérations en conflit et qui sont effectivement exécutées, sont toujours exécutés dans le même ordre
→ pas de dépendances d'orientation opposée → pas de cycle



Verrouillage à deux phases : schéma de preuve

Notation : $e_1 \prec e_2 \equiv$ l'événement e_1 s'est produit avant l'évt. 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 : remarques

Mise en œuvre simple : verrouillage à deux phases strict

- 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 (verrous conservés jusqu'à la fin)

Emploi de verrous

→ restriction du parallélisme potentiel

→ restriction accrue par le report des libérations jusqu'à l'instant du point de verrouillage maximal.

→ risque d'interblocage

Verrouillage à deux phases : traitement de l'interblocage

Techniques classiques
- délai de garde (Tandem...)
- ordre sur la prise des verrous (classes ordonnées)
- prédéclaration (et prise atomique) de *tous* les verrous requis

Techniques particulières : utilisation des estampilles
pour prévenir la formation de cycles dans le graphe d'attente
- $T_i.E$ désigne l'estampille de T_i
- situation : T_i demande l'accès à un objet déjà alloué à T_j
- **wait-die** : si $T_i.E < T_j.E$, bloquer T_i, sinon abandonner T_i
  - attentes permises seulement dans l'ordre des estampilles
  - non préemptif
- **wound-wait** : si $T_i.E < T_j.E$, abandonner T_j, sinon bloquer T_i
  - attentes seulement dans l'ordre inverse des estampilles
  - préemptif ; équitable
  - amélioration : marquer T_j comme « blessée » et attendre qu'elle rencontre un second conflit pour l'abandonner

Conclusion (méthodes de CC) : comment garantir la cohérence *efficacement* ?

Objectif

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

- *Evaluation épisodique*/périodique (après un ensemble de pas)
  → 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)
  → notions de sérialisabilité et de conflit
- *Relâcher* les exigences de cohérence, pour simplifier l'évaluation

  Exemple (BD) : SQL définit quatre niveaux d'isolation
  - *Serializable* : sérialisabilité proprement dite
  - *Repeatable_read* : possibilité de *lectures fantômes*
    (lorsqu'une transaction lit un *ensemble* de données la stabilité de cet ensemble n'est pas garantie : des éléments peuvent apparaître ou disparaître)
  - *Read_commited* : possibilité de lectures fantômes ou *non répétables*
    (la même donnée lue 2 fois de suite peut retourner 2 valeurs différentes)
  - *Read_uncommited* : possibilité de lectures fantômes, non répétables ou *sales*
    (lecture de données écrites par des transactions non validées)

Conclusion (2/2) : bilan sur les méthodes de contrôle de concurrence

- 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

Plan

1 Transactions
- Concurrence et cohérence
- Service transactionnel

2 Atomicité

3 Isolation : contrôle de concurrence
- Principe
- Modélisation
- Résultat fondamental
- Méthodes de contrôle de concurrence
- Méthodes optimistes : certification
- Méthodes pessimistes

4 Mémoire transactionnelle
- Motivation
- Intégration aux langages de programmation
- Réalisation
- Questions ouvertes

Mémoire transactionnelle

But

Fournir un service de contrôle de l'accès concurrent à une mémoire partagée garantissant l'exécution atomique d'une série d'opérations

- niveau matériel : accès à une mémoire/un cache partagé sur un multiprocesseur/multicœur
- niveau logiciel : mécanisme (et service) de contrôle de concurrence des threads d'une application parallèle

Similitudes avec les bases de données

- situation : concurrence d'accès à des données partagées
  → système ouvert
- relation naturelle entre sérialisabilité et exclusion mutuelle : recherche/mise en œuvre d'une cohérence forte

Intérêt

Abstraction

gestion déclarative et automatique de la concurrence

→ élimine les risques d'erreur dans la programmation de la synchronisation : granularité des objets verrouillés, interblocage ; gestion des traitements en attente (ordonnancement, priorité, équité)

Compositionnalité

- l'exécution des transactions est indépendante : il est possible de lancer une nouvelle transaction à tout moment
  → adapté à un environnement ouvert, où l'ensemble des traitements exécutés évolue n'est pas connu à l'avance
- alors que la bonne utilisation des verrous dépend du comportement des autres traitements
  Exemple : prévention de l'interblocage

Intérêt (2)

Exécution spéculative

les protocoles optimistes de CC éliminent les blocages et accroissent donc le parallélisme *potentiel*.

- Réalisation simple de structures de données concurrentes non bloquantes. (Algorithmique très complexe sans transactions)
- Traitement efficace de volumes importants de données irrégulières/évolutives
  - parcours de graphes (sans transactions : algorithmique complexe ou verrou global)
  - simulation, jeux en réseau (évite un calcul préalable pour déterminer les objets voisins à verrouiller)

Remarque : il reste tout à fait possible de faire des erreurs de programmation : transactions trop longues (risque accru d'abandon), ou trop courtes (risque de mauvaise isolation)...

Intégration aux langages de programmation

Interface explicite 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 »

Bloc atomique (mot-clé atomically)

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

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

Application phare : « ellipse » des verrous

Idée

transformer les programmes existants
en remplaçant les sections critiques par des transactions

- motivation : les verrous sont pessimistes.
  → si les conflits sont peu nombreux (ce qui est courant), on réduit inutilement le degré de parallélisme
- expérimentation (Herlihy) sur une JVM implantant (de manière transparente) les blocs synchronized par des transactions.
  Résultats conformes aux prévisions : quelques applications nettement accélérées (facteur 5), une grande majorité modérément accélérées, quelques applications très ralenties (conflits nombreux)
- l'ellipse de verrous a des limites dans de nombreux cas : conflits fréquents, volumes mémoires importants (hors cache)

Poursuite de l'idée de l'ellipse de verrous : transactions et synchronisation

Traitement des conflits

- par la synchronisation → blocage (interactions explicites)
- par les transactions → annulation (interactions transparentes)

→ traduction de la synchronisation dans les transactions

1. Abandonner la transaction en cas de conflit
2. Attendre que des valeurs lues aient changé
3. Relancer (automatiquement) la transaction

→ opération retry

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

Remarque : schéma a priori inefficace et peu pertinent, en général...

Mise en œuvre : spécificités de la mémoire transactionnelle

Par rapport aux transactions « classiques » :
ordres de grandeur différents dans le nombre d'objets, de conflits et dans les temps d'accès

→ recherche d'efficacité :

- utilisation de protocoles simples
- utilisation de la propagation directe (éviter des recopies)
  - → réalisation du contrôle de concurrence plus complexe
    - nécessité de contrôler la propagation des valeurs et d'éviter les effets de bord des transactions annulées
    - → notion d'opacité : sérialisabilité + pas de dépendance par rapport aux transactions actives

MTL : Mémoire transactionnelle logicielle (STM)

Implantation purement logicielle de la mémoire transactionnelle.

Interface explicite

- Opérations sur les transactions : `Start()`, `Commit()`, `Abort()`
- Opérations sur les mots mémoire : `Read(Tx)`, `Write(Tx, val)`

Programmation explicite, ou insertion par le compilateur.

Exemple (1/3)

- Mémoire partagée = tableau Mem[0..Max] de mots mémoire
- Utilisation d'un service de verrous non bloquants, fournissant :
  - L.trylock_shared() demande L en mode partagé→ ok/échec
  - L.trylock() demande L en mode exclusif → ok/échec
  - L.unlock() libère L.
  - un verrou est associé à chaque mot mémoire
    → tableau global L[0..Max] de verrous

Exemple (2/3) : opérations sur la mémoire

Structures de données locales à chaque transaction T_k

- $SvMem[0..Max]$: valeur la mémoire avant T_k
- ensembles *lus*, *ecrits* : indices des mots accédés par T_k

Opérations

- $m.read(T_k)$

```
if m ∉ Tk.lus ∪ Tk.ecrits then
    if not L[m].trylock_shared() then abort(Tk);return "echec";endif;
    Tk.lus := Tk.lus ∪ {m} ;
endif
return Mem[m].read();
```

- $m.write(T_k, val)$

```
if m ∉ Tk.ecrits then
    if not L[m].trylock then abort(Tk); return "echec"; endif;
    Tk.ecrits := Tk.ecrits ∪ {m} ;
    Tk.SvMem[m] := Mem[m].read() ;
endif
Mem[m].write(val);
return "ok";
```

Exemple (3/3) : opérations sur les transactions

- *commit*(T_k)

  unlock_all(T_k);
  return "*ok*" ;

- *abort*(T_k)

  // *restaurer les valeurs ecrites*
  foreach $m \in T_k.ecrits$ do $Mem[m].write(T_k.SvMem[m])$;
  unlock_all(T_k);
  return "*ok*" ;

- *unlock_all*(T)

  // *liberer tous les verrous obtenus par* T
  foreach $m \in T.lus \cup T.ecrits$ do $L[m].unlock()$;
  $T.lus := \emptyset$;
  $T.ecrits := \emptyset$;
  return ;

MTM : Mémoire transactionnelle matérielle (HTM)

Instructions processeur

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

Accès implicite ⇒
code existant automatiquement pris en compte + isolation forte

Implantation

- *ensembles lus/écrits* : pratiquement le rôle du cache
- détection des conflits ≈ cohérence des caches
- *journal avant/après* : dupliquer le cache

MTM – limites

Basées sur l'utilisation des caches mémoire
→

- 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 ↔ ligne de cache
- code non portable (lié à un matériel donné)

Difficultés du modèle : quelques exemples (1/3)

Considérer les conflits avec les transactions validées (sérialisabilité) ou avec toutes (opacité) ?

Propagation directe ⇒ opacité

```
                    init x=y
atomic {                        | atomic {
    if (x != y)                 |     x++; /*(1)*/
        while (true) {}         |     y++; /*(4)*/
}                               | }
```

Difficultés (2/3) : interaction avec code non transactionnel

Lectures non répétables

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

Lectures sales : écritures abandonnées mais observées

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

→ garantir la cohérence ⇐ abandon si conflit hors transaction

Difficultés (3/3) : actions non annulables

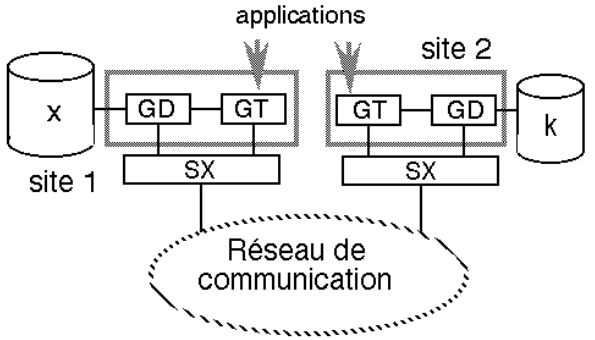
Une transaction annulée doit être sans effet : comment faire s'il y a des effets de bords (p.e. entrées/sorties), avec un contrôle de concurrence optimiste ?

1. Interdire : uniquement des lectures/écritures de variables.
2. 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.
3. virtualiser les actions irrévocables, pour les effectuer seulement après validation

Mémoire transactionnelle : conclusion

+ simple à appréhender
+ ellipse de verrous
+ réduction des erreurs de programmation
+ nombreuses implantations portables en logiciel
  - Java/C++/etc (externe au langage) : XSTM, Deuce, Multiverse
  - Clojure (langage fonctionnel compilé pour la JVM)
  - Haskell (langage fonctionnel)
– surcoût d'exécution, mais
  - la MT logicielle permet de tirer parti des multicœurs, → justifie un surcoût, même important
  - la MT logicielle peut être améliorée (p. ex. couplage avec les mécanismes de la MT matérielle)
– nombreuses sémantiques, souvent floues (mais ce n'est pas pire que les modèles de mémoire partagée)
– questions ouvertes : composition avec le code hors transaction, intégration de la synchronisation

Annexe : architecture de principe du service d'accès aux données



- le *noyau transactionnel* (GT) ordonnance et contrôle les accès aux données de manière à garantir l'atomicité et l'isolation. Les opérations d'accès aux données permises sont transmises
- au *gérant de données* (GD) (SGF ou SGBD) qui réalise les opérations d'accès aux données proprement dites (*traite les requêtes*, dans la terminologie BD)

Neuvième partie

Synchronisation non bloquante

Plan

1 Objectifs et principes
2 Splitter
3 Liste chaînée
4 Conclusion

Objectifs de la synchronisation non bloquante

Problème

Garantir la cohérence d'accès à un objet partagé sans blocage

- Résistance à l'arrêt (crash) d'un processus : un processus donné n'est jamais empêché de progresser, quel que soit le comportement des autres processus
- Vitesse de progression indépendante des autres processus
- Passage à l'échelle
- Surcoût négligeable de synchronisation en cas d'absence de conflit (notion de *fast path*)

Principes généraux

Mécanismes (matériels) utilisés

- registres : protocoles permettant d'abstraire la gestion de la concurrence d'accès à la mémoire partagée (caches...).
  - registres sûrs : toute lecture fournit une valeur écrite ou en cours d'écriture
  - registres réguliers : toute lecture fournit la dernière valeur écrite ou une valeur en cours d'écriture
  - registres atomiques : toute lecture fournit la dernière valeur écrite
- instructions processeur atomiques combinant lecture(s) et écriture(s) (exemple : *TAS*)

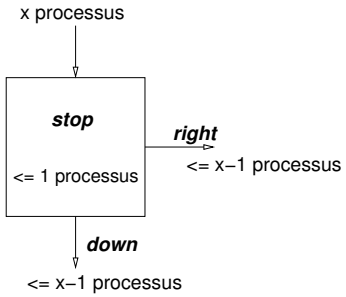
Principes

- chaque processus travaille à partir d'une copie locale de l'objet partagé
- un conflit est détecté lorsque la copie diffère de l'original
- boucle active en cas de conflit d'accès non résolu
  → limiter le plus possible la zone de conflit
- entraide : si un conflit est détecté, un processus peut exécuter des opérations pour le compte d'un autre processus

Plan

1 Objectifs et principes
2 Splitter
3 Liste chaînée
4 Conclusion

Splitter



- x (indéterminé) processus appellent concurremment (ou pas) le splitter
- au plus un processus termine avec *stop*
- si $x = 1$, le processus termine avec *stop*
- au plus $(x - 1)$ processus terminent avec *right*
- au plus $(x - 1)$ processus terminent avec *down*

Splitter

Implantation non bloquante

Deux registres atomiques partagés :
Dernier (init $\perp$) et *PF* (init *false*) // *PF : Porte Fermée*
Chaque processus a un identifiant unique id_i.

direction(id_i)
  Dernier $\leftarrow id_i$;
  if *PF* **then** $dir_i \leftarrow$ *right*;
  else *PF* $\leftarrow$ true;
    if (Dernier $= id_i$) **then** $dir_i \leftarrow$ *stop*;
    else $dir_i \leftarrow$ *down*; **endif**
  endif
  return dir_i;

Schéma de preuve

Validité les seules valeurs retournées sont right, stop et down.

Vivacité ni boucle ni blocage

stop si $x = 1$ évident (un seul processus exécute *direction()*)

au plus $x - 1$ right les processus obtenant right trouvent PF, qui a nécessairement été positionné par un processus obtenant down ou stop

au plus $x - 1$ down soit p_i le dernier processus ayant écrit *Dernier*. Si p_i trouve *PF*, il obtiendra right. Sinon son test *Dernier* $= id_i$ lui fera obtenir stop.

au plus 1 stop soit p_i le *premier* processus trouvant *Dernier* $= id_i$. Alors aucun processus n'a modifié *Dernier* depuis que p_i l'a fait. Donc tous les processus suivants trouveront *PF* et obtiendront right (car p_i a positionné *PF*), et les processus en cours qui n'ont pas trouvé *PF* ont vu leur écriture de *Dernier* écrasée par p_i (puisqu'elle n'a pas changé jusqu'au test par p_i). Ils ne pourront donc trouver *Dernier* égal à leur identifiant et obtiendront donc down.

Application : réétiquetage non bloquant de processus

- Soit n processus d'identité $id_1, \ldots, id_n \in [0..N]$ où $N \gg n$
- On souhaite renommer les processus pour qu'ils aient une identité prise dans [0..M] où $M \ll N$
- Deux processus ne doivent pas avoir la même identité

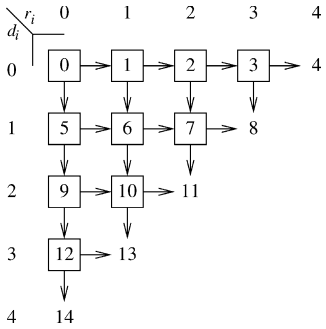
Solution à base de verrous

- Distributeur de numéro accédé en exclusion mutuelle
- $M = n$
- Complexité temporelle : O(1) pour un numéro, O(n) pour tous

Solution non bloquante

- Grille de splitters
- $M = \frac{n(n+1)}{2}$
- Complexité temporelle : O(n) pour un numéro, O(n) pour tous

Grille de splitters



Vivacité : - traversée d'un nombre fini de splitters.
- chaque splitter est non bloquant

Etiquettes uniques : un splitter renvoie stop à un processus au plus

Tout processus obtient une étiquette : - stop si $x = 1$,
- un splitter ne peut orienter tous les processus sur une même direction,
- les bords de la grille sont à distance $n-1$ de l'origine

Algorithme : get_name(id_i)

$d_i \leftarrow 0$; $r_i \leftarrow 0$; $term_i \leftarrow false$;
while $((d_i + r_i < n-1) \wedge \neg term_i)$ **do**
  $Dernier[d_i, r_i] \leftarrow id_i$;
  if $PF[d_i, r_i]$ **then** $r_i \leftarrow r_i + 1$; % *right*
  else $PF[d_i, r_i] \leftarrow true$;
    if $(Dernier[d_i, r_i] = id_i)$ **then** $term_i \leftarrow true$; % *stop*
    else $d_i \leftarrow d_i + 1$; % *down*
    endif
  endif
endwhile
return $(n * d_i + r_i - (d_i(d_i-1)/2))$
  % *le nom en position* d_i, r_i *de la grille*

Plan

1 Objectifs et principes
2 Splitter
3 Liste chaînée
4 Conclusion

File

```
class Nœud<T> { Nœud<T> suiv ; T item ; }
class File<T> {
   Nœud<T> tête ;
   Nœud<T> queue ;
   File() { // Nœud sentinelle en tête
            tête = queue = new Nœud<T>() ;
   }
  void enfiler (T item) {
     Nœud<T> n = new Nœud<T>() ;
     n.item = item ;
     queue.suiv = n ;
     queue = n ;
  }
```

```
T défiler () {
   T rés = null ;
   if (tête != queue) then
       tête = tête.suiv ;
       rés = tête.item ;
   endif
   return rés ;
}
} //File<T>
```

Synchronisation classique : file avec verrou

Conflits

- enfiler/enfiler (queue utilisé en deux endroits)
- défiler/défiler (tête utilisé en deux endroits)
- enfiler/défiler (file avec 1 seul élément)
  - ⇒ tout en exclusion mutuelle

```
void enfiler (T item) {
   Nœud<T> n = new Nœud<T>() ;
   n.item = item ;
   verrou.lock() ;
   queue.suiv = n ;
   queue = n ;
   verrou.unlock() ;
}
```

```
T défiler () {
   T rés = null ;
   verrou.lock() ;
   if (tête != queue) then
       tête = tête.suiv ;
       rés = tête.item ;
   endif
   verrou.unlock() ;
   return rés ;
}
```

- Bloquant définitivement si un processus s'arrête en plein milieu
- Tous les processus sont ralentis par un unique lent
- Compétition systématique enfiler/défiler

Compare-and-set

- Instruction atomique
- Affectation d'une valeur (new),
  à condition que la valeur courante soit égale à (init)

```
boolean CAS(*v, init, new) {
    atomically { if (*v == init) then
                    *v = new;
                    return true;
                 else return false;
                 endif ;
    }
}
```

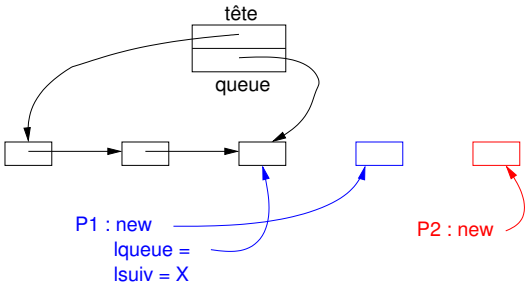
Schéma d'usage

```
*v = init; // copie de init dans *v
opérations concurrentes ;
if CAS(*v, init, new) then
  ok  pas de changement = pas de conflits → affecter et finir
else
    traiter les conflits   → reprendre
endif
```

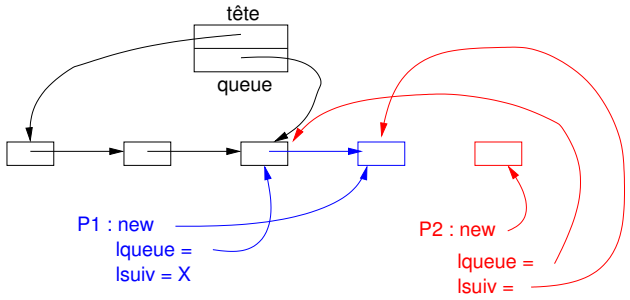
enfiler non bloquant

```
Nœud<T> n = new Nœud<T>;
n.item = item;
loop
  Nœud<T> lqueue = queue;
  Nœud<T> lsuiv = lqueue.suiv;
  if lqueue == queue then      lqueue et lsuiv cohérents ?
    if lsuiv == null then      queue vraiment dernier ?
      if CAS(lqueue.suiv, lsuiv, n) essai lien nouveau nœud
        break;                 succès !
      endif
    else            queue n'était pas le dernier nœud
      CAS(queue, lqueue, lsuiv); entraide : essai m-à-j queue
    endif
  endif
endloop
CAS(queue, lqueue, n); insertion réussie, essai m-à-j queue
```

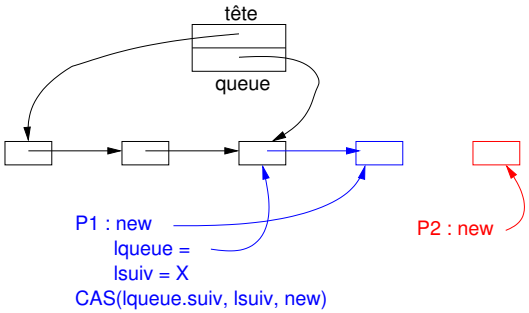
Exemple : deux `enfiler` concurrents



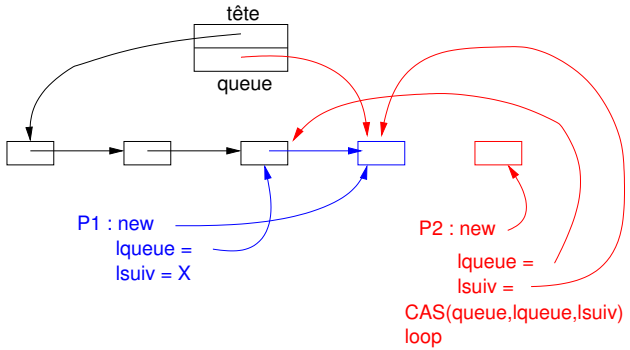
Exemple : deux `enfiler` concurrents



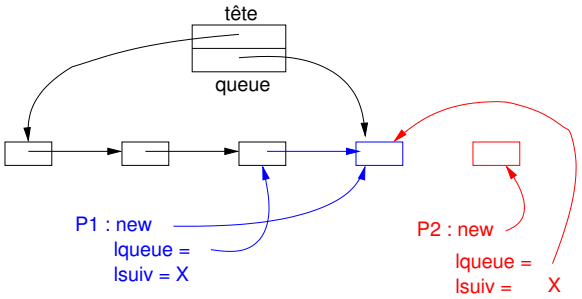
Exemple : deux `enfiler` concurrents



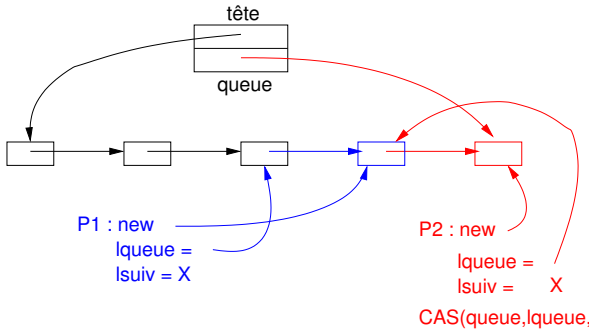
Exemple : deux `enfiler` concurrents



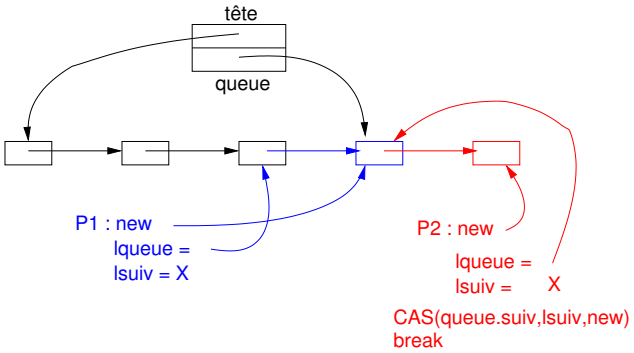
Exemple : deux `enfiler` concurrents



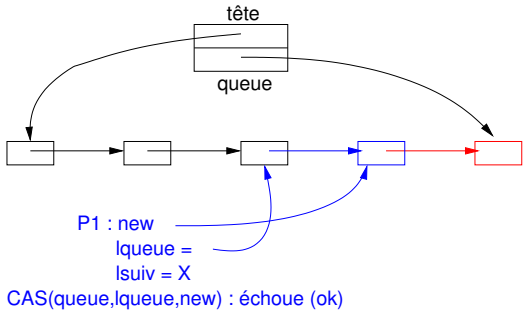
Exemple : deux `enfiler` concurrents



Exemple : deux `enfiler` concurrents



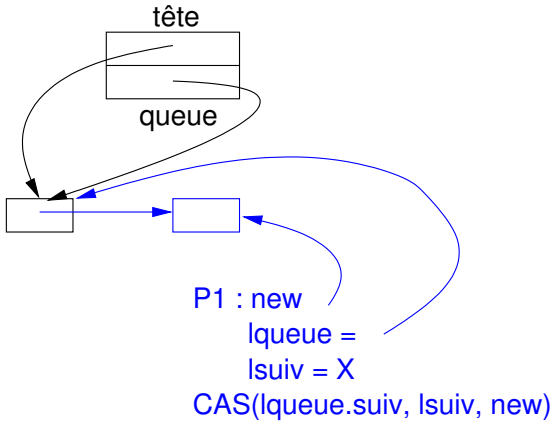
Exemple : deux `enfiler` concurrents



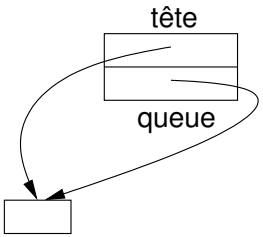
défiler non bloquant

```
loop
  Nœud<T> ltête = tête; Nœud<T> lqueue = queue;
  Nœud<T> lsuiv = ltête.suiv;
  if ltête == tête then  lqueue, ltête, lsuiv cohérents ?
    if ltête == lqueue then  file vide ou queue à la traîne ?
      if (lsuiv == null) then
          return null;       file vide
      endif
       sinon conflit avec enfiler → entraide : essai m-à-j queue
      CAS(queue, lqueue, lsuiv);
    else
      res = lsuiv.item;
      if CAS(tête, ltête, lsuiv) then essai m-à-j tête
          break;                      succès !
      endif
    endif
  endif
endloop sinon (queue ou tête à la traîne) on recommence
return res;
```

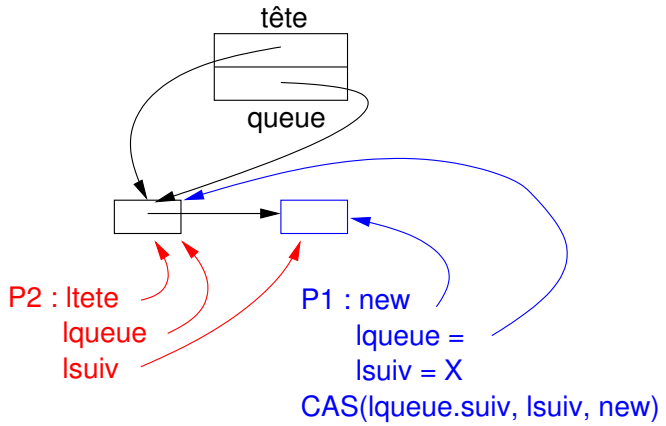

Exemple : défiler et enfiler concurrents



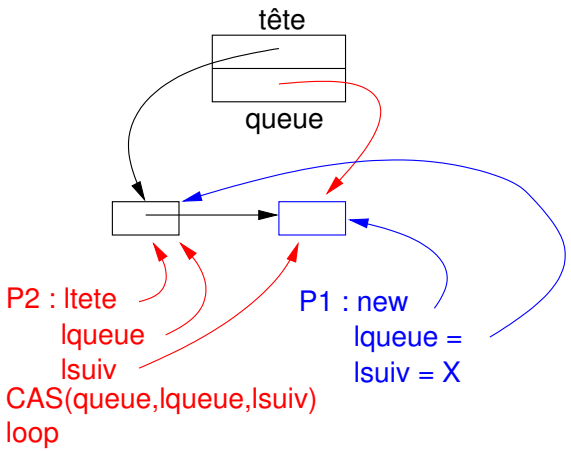
Exemple : défiler et enfiler concurrents



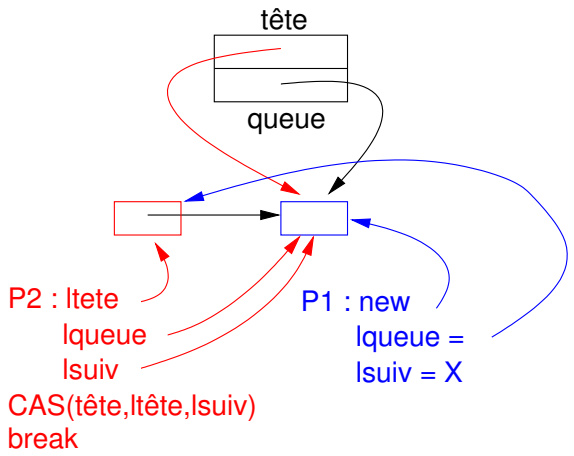
Exemple : défiler et enfiler concurrents



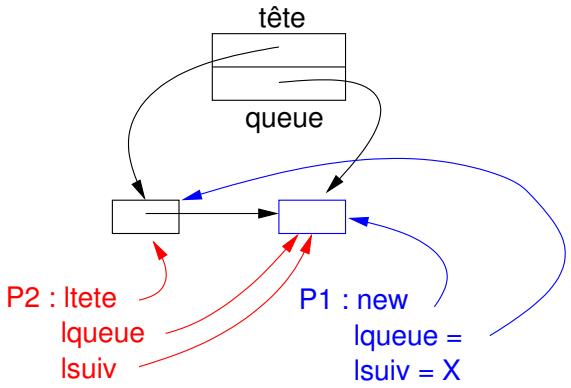
Exemple : `défiler` et `enfiler` concurrents



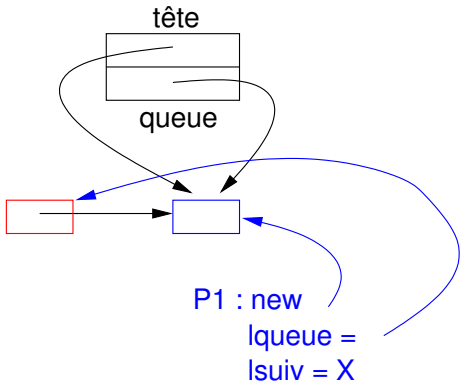
Exemple : `défiler` et `enfiler` concurrents



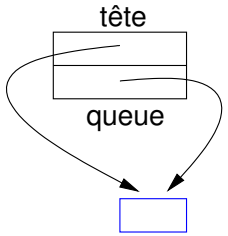
Exemple : `défiler` et `enfiler` concurrents



Exemple : `défiler` et `enfiler` concurrents



Exemple : `défiler` et `enfiler` concurrents



Problème A-B-A

- L'algorithme précédent n'est correct qu'en absence de recyclage des cellules libérées par `défiler`
- Problème A-B-A :
  - P_1 lit x et obtient a ;
  - P_2 change x en b et libère a ;
  - P_3 change x en a ;
  - P_1 effectue CAS(x,a,...), qui réussit et lui laisse croire que x n'a pas changé depuis sa lecture.
- Solutions
  - Compteur de générations, incrémenté à chaque modification $\langle x, \text{gen } i\rangle \neq \langle x, \text{gen } i+2\rangle$
    Nécessite un CAS2(x,a,gen,i,...)
  - Ramasse-miette découplé : retarder la réutilisation d'une cellule (*Hazard pointers*). L'allocation/libération devient alors le facteur limitant de l'algorithme.

Plan

1 Objectifs et principes
2 Splitter
3 Liste chaînée
4 Conclusion

Conclusion

+ performant, même avec beaucoup de processus
+ résistant
– implantation fragile, ad-hoc, peu réutilisable, pas extensible
– implantation complexe
  → nécessité de prouver la correction
– lié à une architecture matérielle
+ bibliothèques spécialisées
  `java.util.concurrent.ConcurrentLinkedQueue`
  `j.u.concurrent.atomic.AtomicReference.compareAndSet`

Conclusion

Programmation parallèle

- souvent utile
- parfois indispensable
- fragile et complexe
- souvent difficile
- stimulant

Deux aspects

- le parallélisme
- la synchronisation

Approches classiques

- création explicite d'activités
- synchronisation explicite
- mécanismes classiques
  (verrou d'exclusion mutuelle, sémaphore, moniteur)
- raisonnablement connues
- schémas classiques
  (producteurs/consommateurs, lecteurs/rédacteurs)

Exemples : Java Thread, C POSIX Threads

Parallélisme et synchronisation gérés

- création implicite d'activités
- synchronisation implicite
- schémas classiques (fork-join)
- encore peu développées
- ne résolvent pas tous les problèmes

Exemple : OpenMP, interface Java `Executor` (pool de threads. . .)

Synchronisation gérée/masquée

- création implicite et explicite d'activités
- synchronisation implicite
  - mémoire transactionnelle
  - bibliothèque de structures de données non bloquantes

Avenir (? ? ?)

- protocoles optimistes/non bloquants
- activités et synchonisation implicites, paresseuses
  → données actives
  langages/modèles fonctionnels : Haskell, Clojure