Systèmes concurrents

2SN

12 septembre 2021

UE Systèmes concurrents et communicants

3 matières

- Systèmes concurrents : modèles, méthodes, outils pour le parallélisme « local »
- Intergiciels : mise en œuvre du parallélisme dans un environnement réparti (machines distantes)
- Projet données réparties : réalisation d'un service de support à la programmation concurrente, parallèle ou répartie.

Evaluation de l'UE

- Examen Systèmes concurrents : écrit, sur la conception de systèmes concurrents
- *(Examen Intergiciels : écrit)*
- Projet commun : réalisation d'un service de support à la programmation concurrente, parallèle ou répartie.
  - présentation mi-octobre, rendu final mi janvier
  - travail en groupe de 4, suivi + points d'étape réguliers

Matière : systèmes concurrents – organisation

Composition

- Cours (50%) : définitions, principes, modèles
- TD (25%) : conception et méthodologie
- TP (25%) : implémentation des schémas et principes

Fonctionnement (si présentiel)

- Cours : classique, avec un soupçon de style classe inversée
  - version sonorisée disponible en ligne
  - pour les séances 6 et 7 : travail en amont de la séance, puis retour et séance en semi-autonomie
- TDs : classique
- TP : classique, avec rendu en fin de semaine

Evaluation

- Si examen sur table : écrit + bonus (rendus TPs, Quiz)
- Si examen à distance : contrôle continu (rendus TPs, quiz) + petit examen en ligne

Pages de l'enseignement : `http://moodle-n7.inp-toulouse.fr`
Contact : `mauran@enseeiht.fr`, `queinnec@enseeiht.fr`

Objectifs

Objectif

Être capable de comprendre et 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 proposant les objets/outils de base

Plan du cours

1. Introduction : problématique
2. Exclusion mutuelle
3. Synchronisation à base de sémaphores
4. Interblocage
5. Synchronisation à base de moniteur
6. API Java, Posix Threads
7. Processus communicants – Go, Ada
8. Transactions – mémoire transactionnelle
9. Synchronisation non bloquante

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

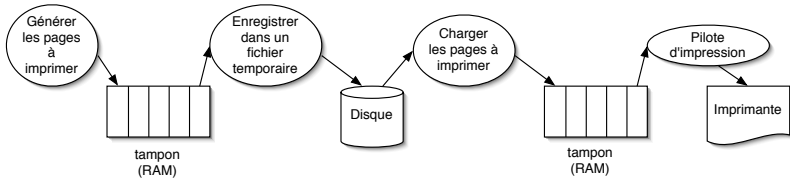
Le problème

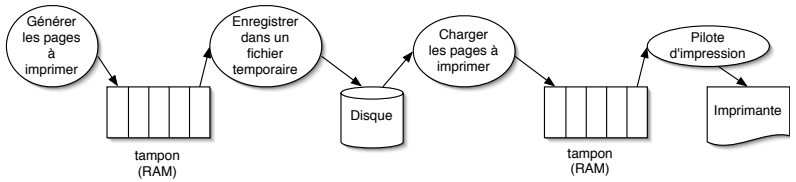
Système concurrent

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

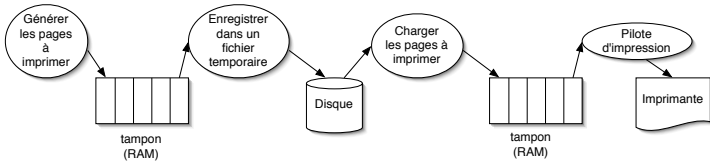




Conception : parallélisation d'un traitement

- 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

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 $b; a$

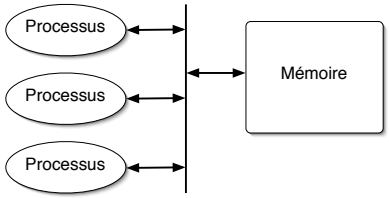
Motivation

- 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

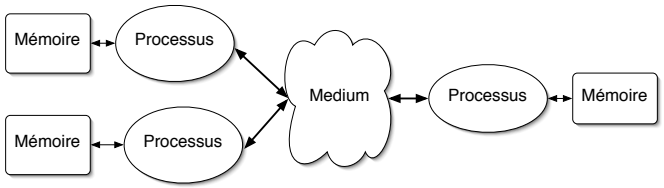
- multiprocesseurs à mémoire partagée,
- processus légers,
- Unix : couplage mémoire (mmap), fichiers

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

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

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

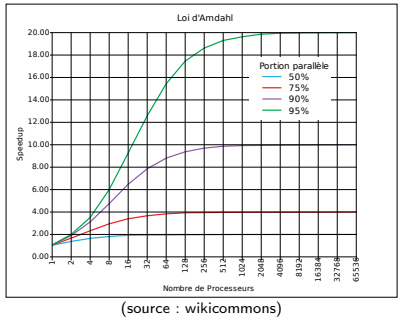
Idée naïve sur le parallélisme

≪ *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

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

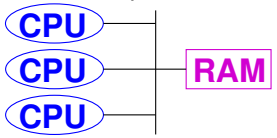
Réalisation sur un monoprocesseur

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)

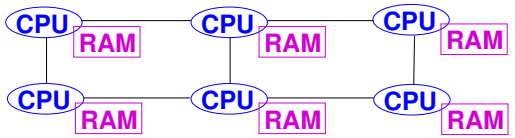
[SMP] Symmetric MultiProcessor :
une mémoire + un ensemble de processeurs



- 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

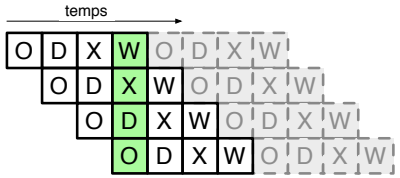
Les architectures récentes éloignent le modèle de la réalité :

- 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

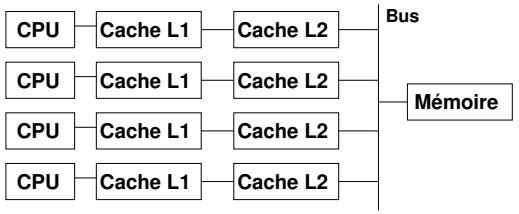
spatiale si on utilise une adresse, on utilisera probablement une adresse proche dans peu de temps

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

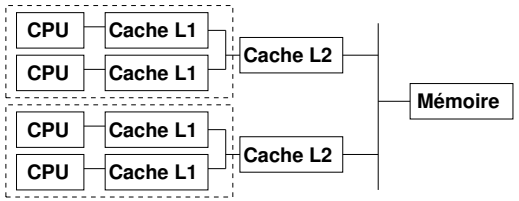
Plusieurs niveaux de caches : de plus en plus gros, de moins en moins rapides (couramment 3 niveaux).

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

L'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

Trop de pain ? ♪♪♪

Vous

1. Arrivez à la maison
2. Constatez qu'il n'y a plus de pain
3. Allez à une boulangerie
4. Achetez du pain
5. Revenez à la maison
6. Rangez le pain

Votre colocataire

1. Arrive à la maison
2. Constate qu'il n'y a plus de pain
3. Va à une boulangerie
4. Achète du pain
5. Revient à la maison
6. Range le pain

Spécification

Propriétés de correction

- Sûreté : un seul pain est acheté
- Vivacité : s'il n'y a pas de pain, quelqu'un en achète

Que se passe-t-il si

- votre colocataire était arrivé après que vous soyez revenu de la boulangerie ?
- Vous étiez arrivé après que votre colocataire soit revenu de la boulangerie ?
- Votre colocataire attend que vous soyez là pour vérifier s'il y a du pain ?

⇒ *race condition* quand la correction dépend de l'ordonnancement des actions

Solution 1 ?

Vous (processus A)

```
A1. si (pas de pain
        && pas de note) alors
A2.   laisser une note
A3.   aller acheter du pain
A4.   enlever la note
   finsi
```

Colocataire (processus B)

```
B1. si (pas de pain)
        && pas de note) alors
B2.   laisser une note
B3.   aller acheter du pain
B4.   enlever la note
    finsi
```

⇒ deux pains possibles si entrelacement A1.B1.A2.B2...

Solution 2 ?

Vous (processus A)

```
laisser une note A
si (pas de note B) alors
   si pas de pain alors
      aller acheter du pain
   finsi
finsi
enlever la note A
```

Colocataire (processus B)

```
laisser une note B
si (pas de note A) alors
   si pas de pain alors
      aller acheter du pain
   finsi
finsi
enlever la note B
```

⇒ zéro pain possible

Solution 3 ?

Vous (processus A)

```
laisser une note A
tant que note B faire
   rien
fintq
si pas de pain alors
     aller acheter du pain
finsi
enlever la note A
```

Colocataire (processus B)

```
laisser une note B
si (pas de note A) alors
   si pas de pain alors
      aller acheter du pain
   finsi
finsi
enlever la note B
```

Pas satisfaisant

Hypothèse de progression / Solution peu évidente / Asymétrique / Attente active

Interférence et isolation

```
(1) x := lire_compte(2);          (a) v := lire_compte(1);
(2) y := lire_compte(1);          (b) v := v - 100;
(3) y := y + x;                   (c) ecrire_compte(1, v);
(4) ecrire_compte(1, y);
```

- Le compte 1 est partagé 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.

Accès concurrents

Exécution concurrente ♪♪♪

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

Modification concurrente ♪♪♪

```
⟨ 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",y,x); ⟩
⇒ affiche 0 0 ou 2 1 ou 0 1 ou 2 0 !
```

Section critique

Définition

Les séquences $S_1 =$ (1) ; (2) ; (3) ; (4) et $S_2 =$ (a) ; (b) ; (c) sont des sections critiques, qui doivent chacune 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 $S_2; S_1$.
- cette équivalence peut être atteinte en contrôlant directement l'ordre d'exécution de S_1 et S_2 (exclusion mutuelle), ou en contrôlant les effets de S_1 et S_2 (contrôle de concurrence).

≪ Y a-t-il du pain ? Si non alors acheter du pain ; ranger le pain. ≫

L'exclusion mutuelle ♪♪♪

Exécution en exclusion mutuelle d'un ensemble de sections critiques

- ensemble d'activités concurrentes A_i
- variables partagées par toutes les activités
  variables privées (locales) à chaque activité
- structure des activités

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

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

Propriétés du protocole d'accès ♪♪♪

- (sûreté) à tout moment, au plus une activité est en cours d'exécution d'une section critique

$$\text{invariant } \forall i, j \in 0..N-1 : A_i.excl \wedge A_j.excl \Rightarrow i = j$$

- (progression ou vivacité globale) lorsqu'il y a (au moins) une demande, une activité qui demande à entrer sera admise

$$(\exists i \in 0..N-1 : A_i.dem) \rightsquigarrow (\exists j \in 0..N-1 : A_j.excl)$$
$$\forall i \in 0..N-1 : A_i.dem \rightsquigarrow (\exists j \in 0..N-1 : A_j.excl)$$

- (vivacité individuelle) si une activité demande à entrer, elle finira par obtenir l'accès (son attente est finie)

$$\forall i \in 0..N-1 : A_i.dem \rightsquigarrow A_i.excl$$

($p \rightsquigarrow q$: à tout moment, si p est vrai, alors q sera vrai ultérieurement)

Plan

Comment ? ♪♪♪

- Solutions logicielles utilisant de l'attente active : tester en permanence la possibilité d'entrer
- Mécanismes matériels
  - simplifiant l'attente active (instructions spécialisées)
  - évitant l'attente active (masquage des interruptions)
- Primitives du système d'exploitation/d'exécution

Forme générale

```
Variables partagées par toutes les activités
Activité Ai
    entrée
    section critique
    sortie
```


Une fausse solution ♪♪♪

Algorithme

```
occupé : shared boolean := false;

tant que occupé faire nop;
occupé ← true;
    section critique
occupé ← false;
```

(Test-and-set non atomique)

Alternance ♪♪♪

Algorithme

```
tour : shared 0..1;

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

- note : **i** = identifiant de l'activité demandeuse
- deux activités (généralisable à plus)
- lectures et écritures atomiques
- alternance obligatoire

Priorité à l'autre demandeur ♪♪♪

Algorithme

```
demande : shared array 0..1 of boolean;

demande[i]  ← true;
tant que demande[j] faire nop;
    section critique
demande[i] ← false;
```

- **i** = identifiant de l'activité demandeuse
  j = identifiant de l'autre activité
- deux activités (non facilement généralisable)
- lectures et écritures atomiques
- risque d'attente infinie (interblocage)

Peterson 1981 ♪♪♪

Algorithme

```
demande: shared array 0..1 of boolean := [false,false];
tour : shared 0..1;

demande[i] ← true;
tour ← j;
tant que (demande[j] et tour = j) faire nop;
    section critique
demande[i] ← false;
```

- deux activités (non facilement généralisable)
- lectures et écritures atomiques
- évaluation non atomique du « et »
- vivacité individuelle

Solution pour *N* activités (Lamport 1974) ♪♪♪

L'algorithme de la boulangerie

```
int num[N];        // numéro du ticket
boolean choix[N];  // en train de déterminer le n°

choix[i] ← true;
int tour ← 0;  // local à l'activité
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) ∧ (num[k],k)≺(num[i],i) faire nop;
section critique
num[i] ← 0;
```

Instruction matérielle TestAndSet ♪♪♪

Retour sur la fausse solution avec test-and-set non atomique de la variable *occupé* (page 17).

Soit TestAndSet(x), instruction indivisible qui positionne x à vrai et renvoie l'ancienne valeur :

Définition

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

Utilisation du TestAndSet

Alors : protocole d'exclusion mutuelle :

Algorithme

```
occupé : shared 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 multiprocesseurs symétriques.

Instruction FetchAndAdd ♪♪♪

Définition

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

```
ticket : shared int := 0;
tour : shared int := 0;
montour : int;   // local à l'activité
montour ← FetchAndAdd(ticket);
tant que tour ≠ montour faire nop;
     section critique
FetchAndAdd(tour);
```

Spinlock x86 ♪♪♪

Spinlock Linux 2.6

```
; initialement Lock = 1
acquire: lock dec word [Lock]
         jns cs              ; jump if not signed
   spin: cmp dword [Lock], 0
         jle spin            ; loop if ≤ 0
         jmp acquire         ; retry entry
     cs:                     ; section critique
release: mov dword [Lock], 1
```

`lock dec` = décrémentation atomique multiprocesseur avec positionnement du bit "sign"

Masquage des interruptions ♪♪♪

Éviter la préemption du processeur par une autre activité :

Algorithme

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

- plus d'attente !
- mono-processeur seulement
- pas d'entrée-sortie, pas de défaut de page, pas de blocage dans la section critique

→ μ-système embarqué

Le système d'exploitation

1. Contrôle de la préemption
2. Contrôle de l'exécution des activités
3. « Effet de bord » d'autres primitives

Ordonnanceur avec priorités ♪♪♪

Ordonnanceur (scheduler) d'activités avec priorité fixe : l'activité de plus forte priorité s'exécute, sans préemption possible.

Algorithme

```
priorité ← priorité max // pas de préemption possible
   section critique
priorité ← priorité habituelle // avec préemption
```

- mono-processeur seulement
- les activités non concernées sont aussi pénalisées
- entrée-sortie ? mémoire virtuelle ?

→ système embarqué

Éviter l'attente active : contrôle des activités ♪♪♪

Algorithme

```
occupé : shared bool := false;
demandeurs : shared fifo;
bloc atomique
     si occupé alors
          self ← identifiant de l´ activité courante
          ajouter self dans demandeurs
          se suspendre
     sinon
          occupé ← true
     finsi
fin bloc
     section critique
bloc atomique
     si demandeurs est non vide alors
          p ← extraire premier de demandeurs
          débloquer p
     sinon
          occupé ← false
     finsi
fin bloc
```

Le système de fichiers (!)

Pour jouer : effet de bord d'une opération du système d'exploitation qui réalise une action atomique analogue au TestAndSet, basée sur l'existence et la création d'un fichier.

Algorithme

```
tant que
    open("toto", O_RDONLY | O_EXCL | O_CREAT, 0) == -1
    // échec si le fichier existe déjà; sinon il est créé
faire nop;
    section critique
unlink("toto");
```

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

La réalité

Actuellement, tout environnement d'exécution fournit un mécanisme de verrou (*lock*), avec les opérations atomiques :

- obtenir (*acquire*) : si le verrou est libre, l'attribuer à l'activité demandeuse ; sinon bloquer l'activité demandeuse
- rendre/libérer (*release*) : si au moins une activité est en attente du verrou, transférer la possession à l'un des demandeurs et le débloquer ; sinon marquer le verrou comme libre.

Algorithme

```
accès : shared lock
accès.acquire
    section critique
accès.release
```

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

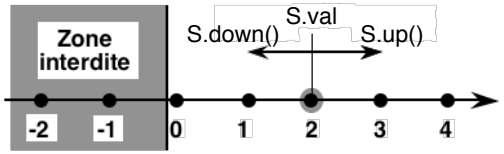
But

- 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 :
  - *down* : bloque si l'état est nul, décrémente l'état s'il est > 0
  - *up* : incrémente l'état
    $\rightarrow$ permet de débloquer un éventuel processus bloqué sur `down`
  - les opérations `down` et `up` sont *atomiques*



- *Autre opération* : constructeur (et/ou initialisation)
  $S = new\ Semaphore(v_0)$ (ou $S.init(v_0)$)
  (crée et) initialise l'état de S à v_0
- *Autres noms des opérations*

P	Probeer (essayer [de passer])	*down*	wait/attendre	acquire/prendre
V	Verhoog (augmenter)	*up*	signal(er)	release/libérer

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.

Définition formelle : Hoare

Définition

Un sémaphore S encapsule un entier val tel que

$$\begin{array}{ccc} \text{init} & \Rightarrow & S.val \geq 0 \\ \{S.val = k \wedge k > 0\} & S.down() & \{S.val = k - 1\} \\ \{S.val = k\} & S.up() & \{S.val = k + 1\} \end{array}$$

Remarques

- Si la précondition de $S.down()$ est fausse, le processus attend.
- Si l'exécution de l'opération *up*, rend vraie la précondition de $S.down()$ et qu'il y a au moins une activité bloquée sur *down*, une telle activité est débloquée (et décrémente le compteur).
- l'invariant du sémaphore peut aussi s'exprimer à partir des nombres $\#down$ et $\#up$ d'opérations *down* et *up* effectuées :
  invariant $S.val = S.val_{\text{init}} + \#up - \#down$

Remarques

1. Lors de l'exécution d'une opération *up*, 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 : *down* non bloquant (*tryDown*)

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

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

Sémaphore binaire (booléen) – Verrou

Définition

Sémaphore S encapsulant un entier b tel que

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

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

Plan

Schémas d'utilisation essentiels (0/4)

Réalisation de l'isolation : exclusion mutuelle

Algorithme

```
global mutex = new Semaphore(--); //objet partagé

// Protocole d'exclusion mutuelle
// (suivi par chacun des processus)

     section critique
```

Schémas d'utilisation essentiels (0/4)

Réalisation de l'isolation : exclusion mutuelle

Algorithme

```
global mutex = new Semaphore(1); //objet partagé

// Protocole d'exclusion mutuelle
// (suivi par chacun des processus)

mutex.down()
     section critique
mutex.up()
```

Schémas d'utilisation essentiels (1/4)

Généralisation : contrôle du degré de parallélisme

Algorithme

Pour limiter à *Max* le nombre d'accès simultanés à la ressource *R* :

- Objet partagé :

```
global accèsR = new Semaphore(Max)
```

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

```
accèsR.down()
     accès à la ressource R
accèsR.up()
```

Règle de conception

- Identifier les portions de code où le parallélisme doit être limité
- Définir un sémaphore pour contrôler le degré de parallélisme
- Encadrer ces portions de code par down/up sur ce sémaphore

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.down()`
- signaler l'occurence de l'événement *E* : `occurrenceE.up()`

Règle de conception

- Identifier les événements qui doivent être attendus avant chaque action
- Définir un sémaphore *semE* par événement *E* à attendre
  - appel à *semE.down()* avant l'action où l'attente est nécessaire
  - appel à *semE.up()* après l'action provoquant l'occurrence de l'événement

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.up()         bArrivé.up()
bArrivé.down()       aArrivé.down()
  ...                  ...
```

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.

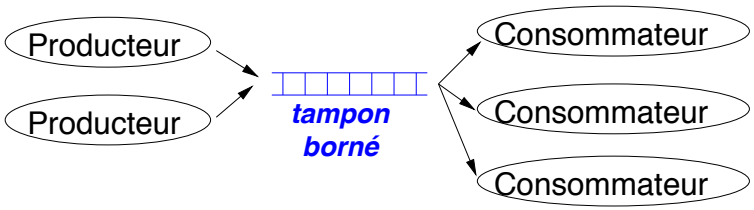
- Objet partagé :

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

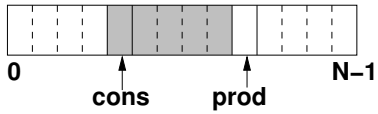
- Protocole de passage de la barrière (pour le processus *i*) :

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


Schéma producteurs/consommateurs : tampon borné



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



producteur	*consommateur*
`produire(i) {i : Item}`	`occupé.down()`
`libre.down()`	*{ ∃ places occupées }*
{ ∃ places libres }	`mutex.down()`
`mutex.down()`	`{ retrait du tampon }`
`{ dépôt dans le tampon }`	`i := tampon[cons]`
`tampon[prod] := i`	`cons := cons + 1 mod N`
`prod := prod + 1 mod N`	`mutex.up()`
`mutex.up()`	*{ ∃ places libres }*
{ ∃ places occupées }	`libre.up()`
`occupé.up()`	`consommer(i) {i : Item}`

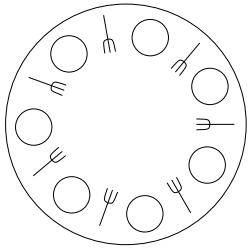
Sémaphores : `mutex` := 1, `occupé` := 0, `libre` := N

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

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.down() =  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.up() =  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) :

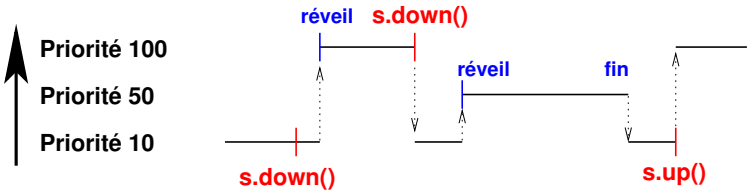
réalisation d'un sémaphore général à partir de sémaphores binaires

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

→ 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 et la validation de solutions

Plan

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

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 $A \rightarrow$ se bloque
4. P_1 demande $B \rightarrow$ se bloque

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 :

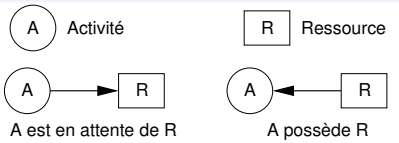
Interblocage $\equiv$ négation de la vivacité faible (progression)

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

Plan

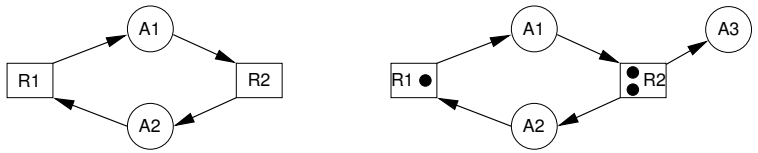
Plan

Notation : graphe d'allocation



Condition nécessaire à l'interblocage

Attente circulaire (cycle dans le graphe d'allocation)



Solutions

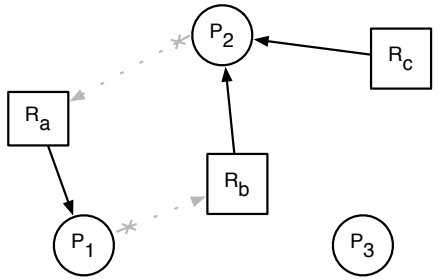
Prévention : empêcher la formation de cycles dans le graphe

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

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

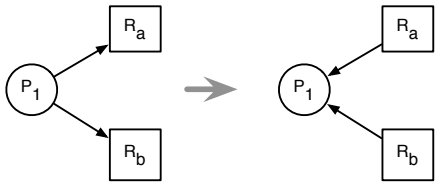
Comment éviter par construction la formation de cycles ? (2/4)

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

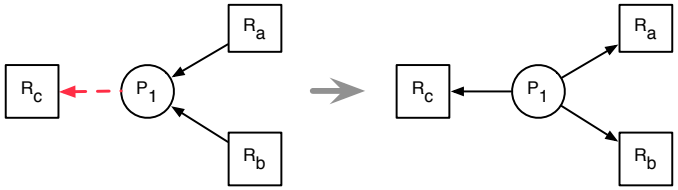
Comment éviter par construction la formation de cycles ? (3/4)

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

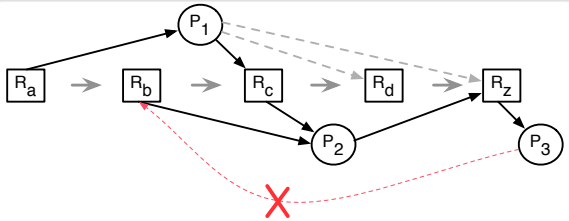
- libérer les ressources qu'il a obtenues
- réobtenir les ressources libérées, avant de pouvoir poursuivre
  → risque de famine



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



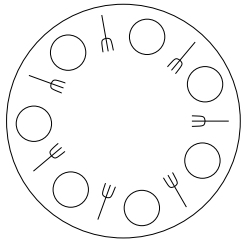
→ pour chaque processus, les chemins du graphe d'allocation vont des ressources inférieures (déjà obtenues) aux supérieures (demandées)

⇒ 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.

Plan

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

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

P_2	9	2	+1	non

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

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;
     fini : booléen = faux;

si Allouées[IdProc]+Demande > Annoncées[IdProc] alors erreur
sinon
   tant que non fini faire
    si Demande > Disponible alors <bloquer le processus>
    sinon
      si étatFiable({1..NbProc}, Disponibles - Demande) alors
        Allouées[IdProc] := Allouées[IdProc] + Demande ;
        Disponibles := Disponibles - Demande;
        fini := vrai;
      sinon <bloquer le processus>;
      finsi
    finsi
  fintq
finsi
```


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

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

- 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

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

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.

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() *[C.wait()]* : bloque et range dans la file associée à C le processus appelant, puis libère l'accès exclusif au moniteur.
  - C.signaler() *[C.signal()]* : 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
- 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

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

Les activités (processus utilisant le moniteur)

Client	Serveur
boucle ⋮ déposer_travail(t) ⋮ r ←lire_résultat() ⋮ *fin_boucle*	*boucle* ⋮ x ← prendre_travail() // ($y \leftarrow f(x)$) rendre_résultat(y) ⋮ *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*
	Dépôt.attendre()
req ← t	*finsi*
	t ← req
	req ← null
Dépôt.signaler()	{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()*,

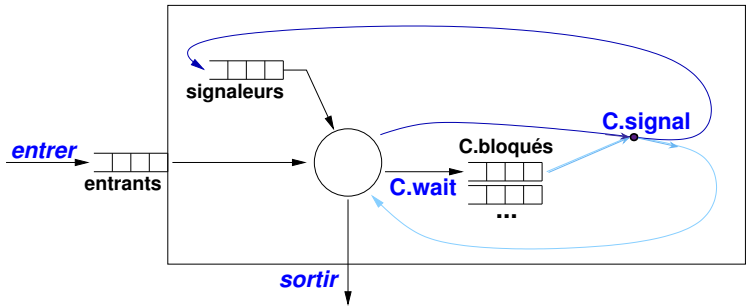
- l'accès exclusif est transféré au processus réveillé (signalé) ;
- le processus signaleur est mis en attente dans une file globale spécifique, prioritaire sur les processus entrants

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.

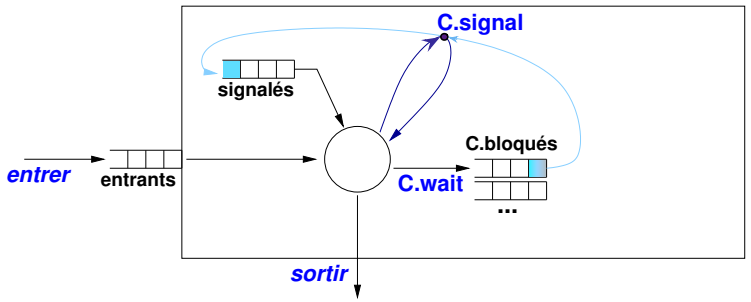
Priorité au signalé



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)

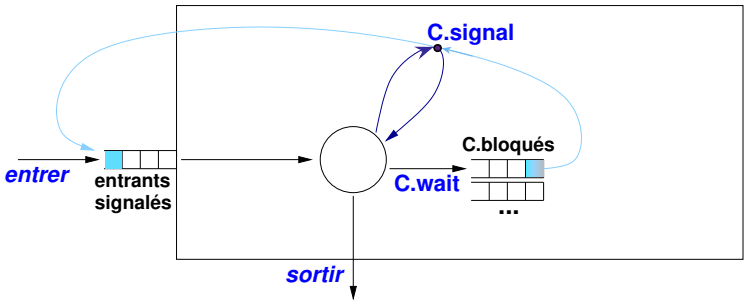
Priorité au signaleur avec file spécifique des signalés



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

Exemple signaleur vs signalé : travail délégué avec 1 client, 2 ouvriers

Priorité au signalé

OK : quand un client dépose une requête et débloque un ouvrier, celui-ci obtient immédiatement l'accès exclusif et prend la requête.

Priorité au signaleur

- KO : situation : ouvrier n°1 bloqué sur `Dépôt.attendre()`.
- Le client appelle `déposer_travail` et en parallèle, l'ouvrier n°2 appelle `prendre_travail`. L'ouvrier n°2 attend l'accès exclusif.
- Lors de `Dépôt.signaler()`, l'ouvrier n°1 est débloqué de la var. condition et se met en attente de l'accès exclusif.
- Quand le client libère l'accès exclusif, qui l'obtient ? Si ouvrier n°2, il « vole » la requête, puis ouvrier n°1 obtient l'accès exclusif et récupère `null`.

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 :

⇒ à chaque nouvel état (= à chaque affectation),
évaluer 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

Exemple – le moniteur (reprise planche 10)

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()

Plan

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

Schéma 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 qui peuvent progresser 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)

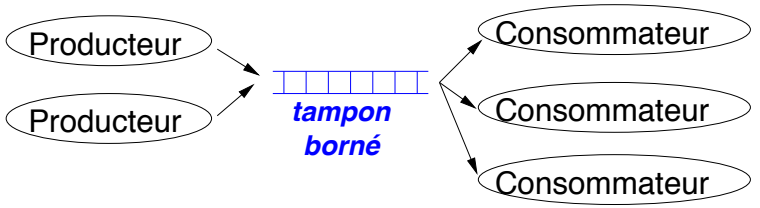
Structure standard d'une opération

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 ¬(nbOccupées < N) alors
     PasPlein.attendre()
finsi
{ (1) nbOccupées < N }
// action applicative (ranger v dans le tampon)
nbOccupées + +
{ (2) N ≥ nbOccupées > 0 }
PasVide.signaler()
```

retirer(out v)

```
si ¬(nbOccupées > 0) alors
     PasVide.attendre()
finsi
{ (3) nbOccupées > 0 }
// action applicative (prendre v dans le tampon)
nbOccupées − −
{ (4) 0 ≤ nbOccupées < N }
PasPlein.signaler()
```

Vérification & Priorité

- Vérification : (2) ⇒ (3) ? (4) ⇒ (1) ?
- Si priorité au signaleur, transformer `si` en `tant que` :

déposer(in v)

```
tant que ¬(nbOccupées < N) faire
     PasPlein.wait
fintq
{ (1) nbOccupées < N }
// action applicative (ranger v dans le tampon)
nbOccupées + +
{ (2) N ≥ nbOccupées > 0 }
PasVide.signal
```

Plan

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

Plan

Allocateur de ressources

- N ressources équivalentes, une activité en demande $p \in 1..N$ puis les libère.
- Bon comportement : pas deux demandes consécutives sans libération (cf interblocage).
- Difficulté : une libération peut débloquer 0, 1 ou plusieurs demandeurs selon le nombre de ressources rendues et attendues.

Allocateur de ressources - méthodologie

1. Interface :
   - `demander(p: 1..N)`
   - `libérer(q: 1..N)`
2. Prédicats d'acceptation :
   - demander(p) : il y a au moins p ressources libres
   - retirer(q) : rien
3. Variables d'état :
   - `nbDispo` : natural
   - demander(p) : `nbDispo` $\geq p$
   - libérer(q) : *true*
4. Invariant : $0 \leq$ `nbDispo` $\leq N$
5. Variable condition : `AssezDeRessources`

Allocateur – opérations

demander(p)

```
si demande ≠ 0 alors -- il y a déjà un demandeur → j'attends mon tour
    Sas.wait
finsi
si ¬(nbDispo < p) alors
    demande ← p
    AssezDeRessources.wait     -- au plus un bloqué ici
    demande ← 0
finsi
nbDispo ← nbDispo − p
Sas.signal    -- au suivant de demander
```

libérer(q)

```
nbDispo ← nbDispo + p
si nbDispo ≥ demande alors
    AssezDeRessources.signal
finsi
```

Note : priorité au signaleur ⇒ transformer le premier "si" de demander en "tant que" (ça suffit ici).

Variante : réveil multiple : signalAll/broadcast

C.signalAll (ou broadcast) : *toutes* les activités bloquées sur la variable condition *C* sont débloquées. Elles se mettent en attente de l'accès exclusif.

Rarement utilisé à bon escient. Une solution triviale à un problème de synchronisation est d'utiliser une *unique* variable condition `Accès` et d'écrire *toutes* les procédures du moniteur sous la forme :

```
tant que ¬(condition d'acceptation) faire
    Accès.wait
fintq
...
Accès.signalAll  -- battez-vous
```

Mauvaise idée ! (performance, prédictibilité)

Réveil multiple : cour de récréation unisexe

0. `type genre` $\triangleq$ `(Fille, Garçon)`
   `inv(g)` $\triangleq$ `si g = Fille alors Garçon sinon Fille`
1. Interface : entrer(genre) / sortir(genre)
2. Prédicats : entrer : personne de l'autre sexe / sortir : –
3. Variables : nb(genre)
4. Invariant : nb(Filles) $= 0 \vee$ nb(Garçons) $= 0$
5. Variables condition : accès(genre)
6.

```
entrer(genre g)                 | sortir(genre g)
  si nb(inv(g)) ≠ 0 alors       |   nb(g)--
     accès(g).wait              |   si nb(g) = 0 alors
  finsi                         |       accès(inv(g)).signalAll
  nb(g)++                       |   finsi
```

(solution naïve : risque de famine si un genre se coalise pour avoir toujours un membre présent dans la cour)

Priorité au signaleur : transformation systématique ?

Pour passer de priorité au signalé à priorité au signaleur, transformer ≪ `si CA` ≫ en ≪ `tant que CA` ≫ n'est correct que si la condition d'acceptation (à l'entrée) et la condition de déblocage (au réveil) sont identiques.
Exemple : évitement de la famine : variable `attente(genre)` pour compter les enfants en attente et ne pas accaparer la cour.

```
entrer(genre g)
  si nb(inv(g)) ≠ 0 ∨ attente(inv(g)) ≥ 4 alors
     attente(g)++
     accès(g).wait
     attente(g)--
  finsi
  nb(g)++
```

Interblocage possible avec priorité signaleur et ≪ tant que ≫ à la place du ≪ si ≫ → repenser la solution.

Variante : 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)
  region
    nbOcc, tampon, dépôt
  when
    nbOcc < N
  do
    tampon[dépôt] ← m
    dépôt ← dépôt + 1 % N
    nbOcc ← nbOcc + 1
  end
```

```
retirer()
  region
    nbOcc, tampon, retrait
  when
    nbOcc > 0
  do
    Result ← tampon[retrait]
    retrait ← retrait + 1 % N
    nbOcc ← nbOcc - 1
  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*

Systèmes concurrents

2SN

ENSEEIHT
Département Sciences du Numérique

22 octobre 2020

Sixième partie

Programmation multiactivité
Java & Posix Threads

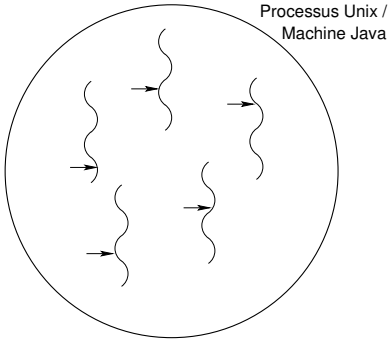
Contenu de cette partie

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

Processus multiactivité

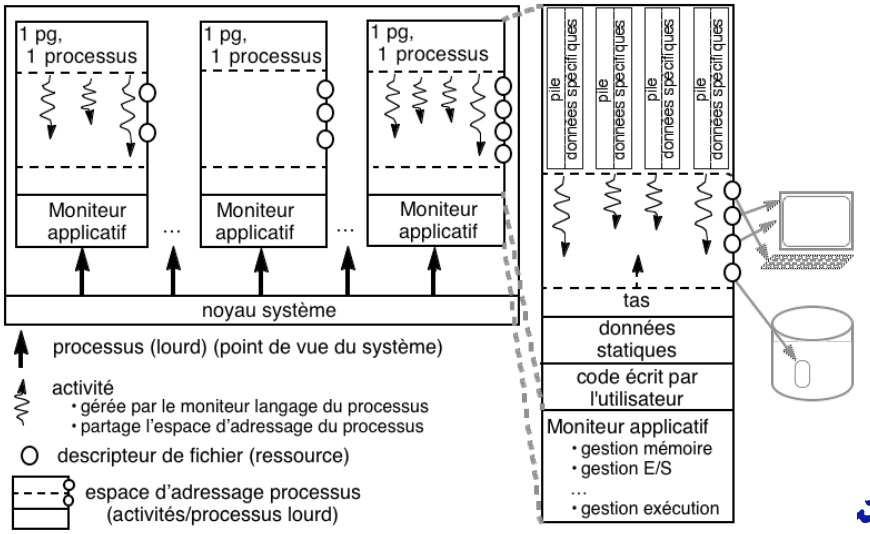


1 espace d'adressage, plusieurs flots de contrôle.
⇒ plusieurs activités (ou processus légers) au sein d'un même processus UNIX / d'une même machine virtuelle Java.

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

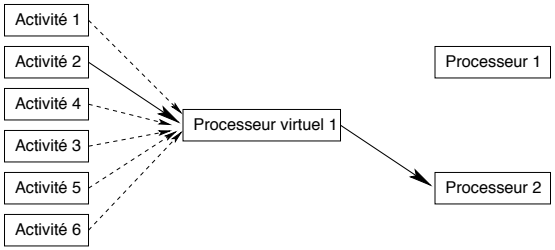
Remarque : la mise en œuvre des processus légers est directe lorsque le système d'exploitation fournit un service d'activités noyau et permet de coupler activités noyau et activités applicatives

Processeurs virtuels

Entre le processeur physique et les activités, il existe généralement une entité interne au noyau, appelé *kernel process* ou *processeur virtuel*.
Cette entité est *généralement* l'unité de blocage : un appel système bloquant (`read...`) bloque le processeur virtuel qui l'exécutait.

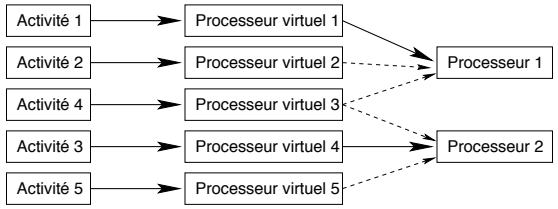
1. Many-to-one : 1 seul processeur virtuel par processus
2. Many-to-many : 1 processeur virtuel par activité
3. Many-to-few : quelques processeurs virtuels par processus

Many-to-one



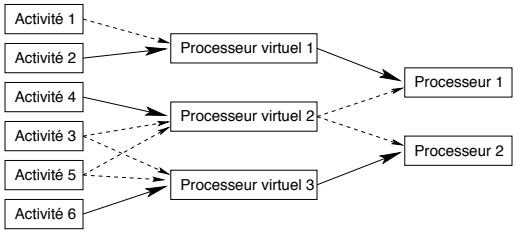
+ commutation entre activités efficace
+ implantation simple et portable
– pas de bénéfice si plusieurs processeurs
– blocage du processus (donc de toutes les activités) en cas d'appel système bloquant, ou implantation complexe

Many-to-many



+ vrai parallélisme si plusieurs processeurs physiques
+ pas de blocage des autres activités en cas d'appel bloquant
– commutation moins efficace (dans le noyau)
– ressources consommées élevées

Many-to-few



+ vrai parallélisme si plusieurs processeurs physiques
+ meilleur temps de commutation
+ meilleur rapport ressources/nombre d'activités
+ pas de blocage des autres activités en cas d'appel bloquant
– complexe, particulièrement si création automatique de nouveaux processeurs virtuels
– faible contrôle des entités noyau

Plan

Conception d'applications parallèles en Java

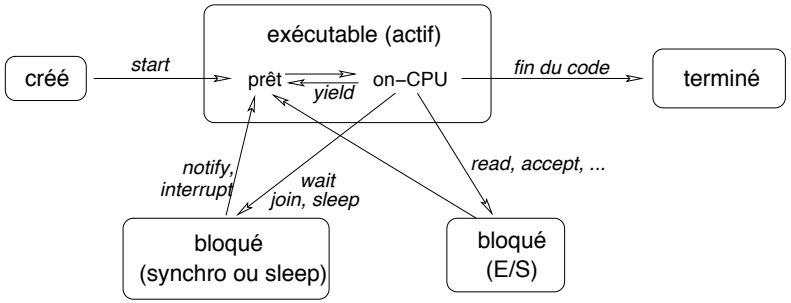
Java permet de manipuler

- 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 : programmation asynchrone/fonctionnelle

Cycle de vie d'une activité



Création d'une activité – interface `Runnable`

Code d'une activité

```
class MonActivité implements Runnable {
    public void run() { /* code de l'activité */ }
}
```

Création d'une activité

```
Runnable a = new MonActivité(...);
Thread t = new Thread(a); // activité créée
t.start(); // activité démarrée
...
t.join(); // attente de la terminaison
```

```
Thread t = new Thread(() -> { /* code de l'activité */ });
t. start ();
```

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 – héritage de `Thread`

Héritage de la classe `Thread` et redéfinition de la méthode `run` :

Définition d'une activité

```
class MonActivité extends Thread {
    public void run() { /* code de l'activité */ }
}
```

Utilisation

```
MonActivité t = new MonActivité(); // activité créée
t.start(); // activité démarrée
...
t.join(); // attente de la terminaison
```

Déconseillé : risque d'erreur de redéfinition de `Thread.run`.

Quelques méthodes

Classe `Thread` :

`static Thread currentThread()`
: obtenir l'activité appelante

`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)

`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)

Interruption

Mécanisme minimal permettant d'interrompre une activité.
La méthode `interrupt()` appliquée à une activité provoque

soit la levée de l'exception `InterruptedException` si l'activité est bloquée sur une opération de synchronisation (`Thread.join`, `Thread.sleep`, `Object.wait...`)

soit le positionnement d'un indicateur `interrupted`, testable :

- `boolean isInterrupted()` qui renvoie la valeur de l'indicateur 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.

Pas d'interruption des entrées-sorties bloquantes ⇒ peu utile.

Données localisées / spécifiques

Un même objet localisé (instance de `InheritableThreadLocal` ou `ThreadLocal`) possède une valeur spécifique dans chaque activité.

```
class MyValue extends ThreadLocal {
    // surcharger éventuellement initValue
}
class Common {
   static MyValue val = new MyValue();
}
// thread t1                    || // thread t2
o = new Integer(1);             || o = "machin";
Common.val.set(o);              || Common.val.set(o);
x = Common.val.get();           || x = Common.val.get();
```

Utilisation ≈ variable locale à chaque activité : identité de l'activité, priorité, date de création, requête traitée...

Plan

Objets de synchronisation

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 autorisant/facilitant les accès concurrents
  - accès atomiques : `ConcurrentHashMap`...
  - accès non bloquants : `ConcurrentLinkedQueue`

Moniteur Java

Principe des moniteurs

- 1 verrou assurant l'exclusion mutuelle
- plusieurs variables conditions associées à ce verrou
- attente/signalement de ces variables conditions
- = un moniteur
- pas de priorité au signalé et pas de file des signalés

Moniteur Java - un producteur/consommateur (1)

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

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


Sémaphores

Sémaphore

```
Semaphore sem = new Semaphore(1); // nb initial de jetons
sem.acquire();                    // = down
sem.release();                    // = up
```

```
public class ProdConSem {
  private Semaphore mutex, placesVides, placesPleines;
  private Object[] items;
  private int depot, retrait;
  public ProdConSem(int nbElems) {
      items = new Object[nbElems];
      depot = retrait = 0;
      placesVides = new Semaphore(nbElems);
      placesPleines = new Semaphore(0);
      mutex = new Semaphore(1);
  }
  ...
```


Moniteur Java - un producteur/consommateur (2)

```
    ...
    public Object retirer() throws InterruptedException {
        verrou.lock();
        while (nbElems == 0)
          pasVide.await();
        Object x = items[retrait];
        retrait = (retrait + 1) % items.length;
        nbElems--;
        pasPlein.signal();
        verrou.unlock();
        return x;
    }
}
```

Sémaphores - un producteur/consommateur (2)

```
    ...
    public void deposer(Object x) throws InterruptedException {
        placesVides.acquire();
        mutex.acquire();
        items[depot] = x;
        depot = (depot + 1) % items.length;
        mutex.release();
        placesPleines.release();
    }

    public Object retirer() throws InterruptedException {
        placesPleines.acquire();
        mutex.acquire();
        Object x = items[retrait];
        retrait = (retrait + 1) % items.length;
        mutex.release();
        placesVides.release();
        return x;
    }
}
```

Producteurs/consommateurs

Paquetage `java.util.concurrent`

BlockingQueue

`BlockingQueue` = producteurs/consommateurs (interface)

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

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

```
BlockingQueue bq = new ArrayBlockingQueue(4); // capacité
bq.put(m);      // dépôt (bloquant) d'un objet en queue
x = bq.take(); // obtention (bloquante) de l'objet en tête
```

Barrière

java.util.concurrent.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(
        () -> { 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.

Compteurs, Verrous L/R

java.util.concurrent.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 `java.util.concurrent.locks.ReadWriteLock`

Verrous pouvant être acquis en mode

- exclusif (`writeLock().lock()`),
- partagé avec les autres non exclusifs (`readLock().lock()`)

→ schéma lecteurs/rédacteurs.

Implantation : `ReentrantReadWriteLock` (avec/sans équité)

Atomicité à grain fin

Outils pour réaliser la coordination par l'accès à des données partagées, plutôt que par suspension/réveil (attente/signal d'événement)

- le paquetage `java.util.concurrent.atomic` fournit des classes qui permettent des accès atomiques cohérents,
- et des opérations de mise à jour conditionnelle du type `TestAndSet`.
- Les lectures et écritures des références déclarées `volatile` sont atomiques et cohérentes.

⇒ synchronisation non bloquante

Danger

Concevoir et valider de tels algorithmes est très ardu. Ceci a motivé la définition d'objets de synchronisation (sémaphores, moniteurs...) et de patrons (producteurs/consommateurs...)

Plan

Services de régulation du parallélisme : exécuteurs

Idée

Séparer la création et la vie des activités des autres aspects (fonctionnels, synchronisation...)
→ définition d'un service de gestion des activités (exécuteur), régulant/adaptant le nombre d'activités effectivement actives, en fonction de la charge courante et du nombre de CPU disponibles :

- trop d'activités → consommation de ressources inutile
- pas assez d'activités → capacité de calcul sous-utilisée

Interfaces d'exécuteurs

- Interface `java.util.concurrent.Executor` :
  `void execute(Runnable r)`,
  - fonctionnellement équivalente à `(new Thread(r)).start()`
  - mais `r` ne sera pas forcément exécuté immédiatement / par une nouvelle activité.
- Interface `java.util.concurrent.ExecutorService` :
  `Future<T> submit(Callable<T> task)`
  soumission d'une tâche rendant un résultat, récupérable ultérieurement, de manière asynchrone.
- L'interface `ScheduledExecutorService` est un `ExecutorService`, avec la possibilité de spécifier un calendrier (départs, périodicité...) pour les tâches exécutées.

Utilisation d'un Executor (sans lambda)

```
import java.util.concurrent.*;
public class ExecutorExampleOld {
 public static void main(String[] a) throws Exception {
   final int NB = 10;
   ExecutorService exec = Executors.newCachedThreadPool();
   Future<?>[] res = new Future<?>[NB];
   for (int i = 0; i < NB; i++) { // lancement des travaux
     int j = i;
     exec.execute(new Runnable() {
                    public void run() {
                        System.out.println("hello" + j); }});
     res[i] = exec.submit(new Callable<Integer>() {
                    public Integer call() { return 3 * j; }});
   }
   // récupération des résultats
   for (int i = 0; i < NB; i++) {
       System.out.println(res[i].get());
   }
 }
}
```

Utilisation d'un Executor (avec lambda)

```
import java.util.concurrent.*;
public class ExecutorExample {
 public static void main(String[] a) throws Exception {
    final int NB = 10;
    ExecutorService exec = Executors.newCachedThreadPool();
    Future<?>[] res = new Future<?>[NB];

    // lancement des travaux
    for (int i = 0; i < NB; i++) {
      int j = i;
      exec.execute(() -> { System.out.println("hello" + j); });
      res[i] = exec.submit(() -> { return 3 * j; });
    }

    // récupération des résultats
    for (int i = 0; i < NB; i++) {
        System.out.println(res[i].get());
    }
 }
}
```

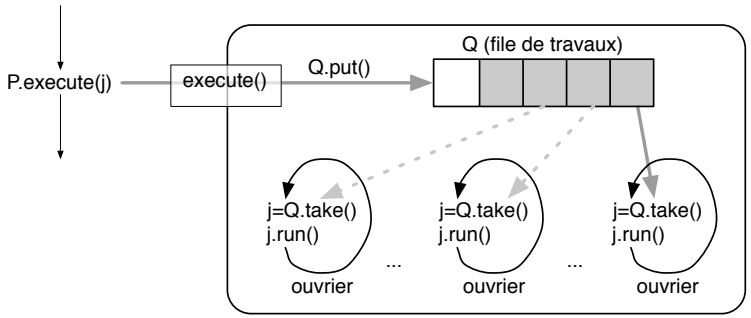
Implantation minimale d'un pool de threads

```
import java.util.concurrent.*;
public class NaiveThreadPool2 implements Executor {
  private BlockingQueue<Runnable> queue;

  public NaiveThreadPool2(int nthr) {
    queue = new LinkedBlockingQueue<Runnable>();
    for (int i=0; i<nthr; i++)
        (new Thread(new Worker())).start();
  }

  public void execute(Runnable job) { queue.put(job); }

  private class Worker implements Runnable {
    public void run() {
      while (true) {
        Runnable job = queue.take(); // bloque si besoin
        job.run();
      }
  }}
}
```


Pool 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)]*

Exécuteurs prédéfinis

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

- Nombre fixe d'activités : `newSingleThreadExecutor()`, `newFixedThreadPool(int nThreads)`
- Nombre d'activités adaptable : `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. 1 min) s'est écoulé, terminaison d'une activité inoccupée
- Parallélisme massif avec vol de jobs : `newWorkStealingPool(int parallelism)`

`java.util.concurrent.ThreadPoolExecutor` permet de contrôler les paramètres de la stratégie d'exécution : politique de la file (FIFO, priorités...), file bornée ou non, nombre minimal / maximal de threads...

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
```

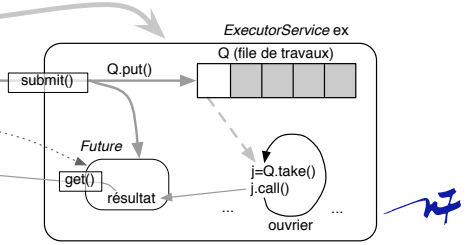
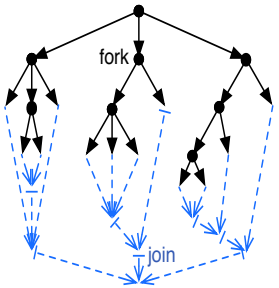


Schéma diviser pour régner (fork/join, map/reduce)

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
  }
```

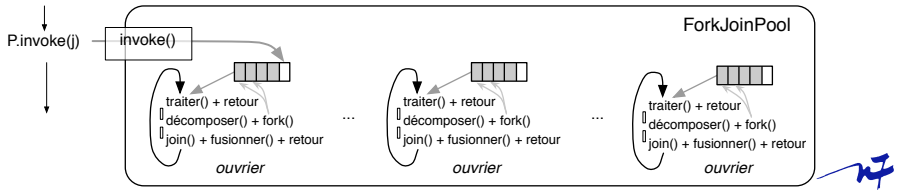


Exécuteur pour le schéma fork/join (1/3)

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)

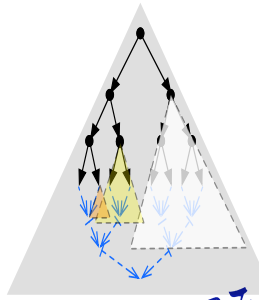
Activité d'un ouvrier du ForkJoinPool :

- 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

→

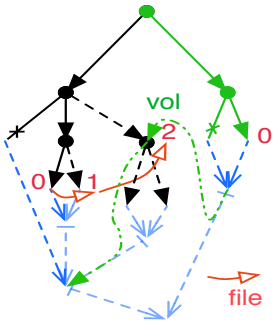
Chaque ouvrier traite un arbre de tâches qu'il

- parcourt et traite en profondeur d'abord → 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)

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.

Plan

Synchronisation (Java ancien)

Obsolète

La protection par exclusion mutuelle (`synchronized`) sert encore, mais éviter la synchronisation sur objet et préférer les véritables moniteurs introduits dans Java 5.

Principe

- exclusion mutuelle
- attente/signalement sur un objet
- équivalent à un moniteur avec une seule variable condition

Exclusion mutuelle

Tout objet Java est équipé d'un verrou d'exclusion mutuelle.

Code synchronisé

```
synchronized (unObj) {
    // Exclusion mutuelle vis-à-vis des autres
    // blocs synchronized(cet objet)
}
```

Méthode synchronisée

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

Équivalent à :

T uneMethode(...) { **synchronized** (**this**) { ... } }

(exclusion d'accès à l'objet sur lequel on applique la méthode, pas à la méthode elle-même)

Exclusion mutuelle

Chaque classe possède aussi un verrou exclusif qui s'applique aux méthodes de classe (méthodes statiques) :

```
class X {
    static synchronized T foo() { ... }
    static synchronized T' bar() { ... }
}
```

`synchronized` assure l'exécution en exclusion mutuelle pour toutes les méthodes statiques synchronisées de la classe `X`.
Ce verrou ne concerne pas l'exécution des méthodes d'objets.

Synchronisation par objet

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 unique activité bloquée sur l'objet, et la met en attente de l'obtention de l'accès exclusif (si aucune activité n'est bloquée, l'appel ne fait rien) ;

`unObj.notifyAll()` réveille toutes les activités bloquées sur l'objet, qui se mettent toutes en attente de l'accès exclusif.

Synchronisation basique – exemple

```
class StationVeloToulouse {
    private int nbVelos = 0;

    public void prendre() throws InterruptedException {
        synchronized(this) {
            while (this.nbVelos == 0) {
                this.wait();
            }
            this.nbVelos--;
        }
    }

    public void rendre() {
        // assume : toujours de la place
        synchronized(this) {
            this.nbVelos++;
            this.notify();
        }
    }
}
```

Synchronisation basique – exemple

```
class BarriereBasique {
    private final int N;
    private int nb = 0;
    private boolean ouverte = false;
    public BarriereBasique(int N) { this.N = N; }

    public void franchir() throws InterruptedException {
        synchronized(this) {
            this.nb++;
            this.ouverte = (this.nb >= N);
            while (! this.ouverte)
              this.wait();
            this.nb--;
            this.notifyAll();
        }
    }
    public synchronized void fermer() {
        if (this.nb == 0)
          this.ouverte = false;
    }
}
```

Difficultés

- prises multiples de verrous :

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

  o1 est libéré par `wait`, mais pas o2

- une seule notification possible pour une exclusion mutuelle donnée → résolution difficile des problèmes de synchronisation

Pas des moniteurs de Hoare !

- programmer comme avec des sémaphores
- affecter un objet de blocage distinct à chaque requête et gérer soit-même les files d'attente

- pas de priorité au signalé, pas d'ordonnancement sur les déblocages


```
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();
       }
     }
   }
}
```

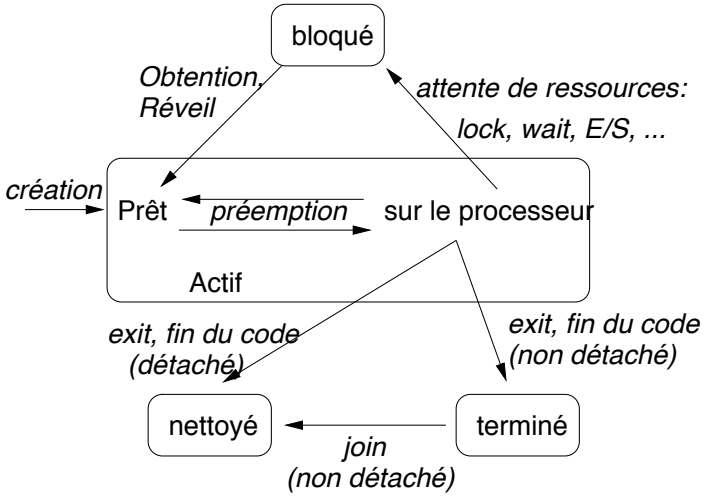
Plan

Posix Threads

Standard de librairie multiactivité pour le C, supporté par de nombreuses implantations plus ou moins conformantes.
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.

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 thread);
```

Détache l'activité `thread`.
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 process 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`.

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);
```

Synchronisation PThread

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 reobtenu 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.

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 (ouf).

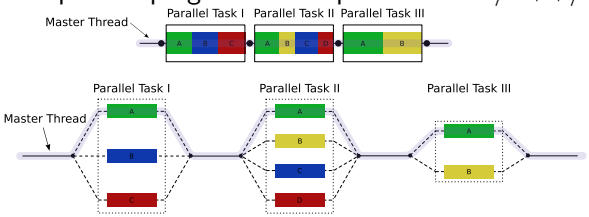
.NET (C#)

Très similaire à Java ancien :

- 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 multiprocesseur à mémoire partagée
– compilateur dédié
– peu de primitives de synchronisation (atomicité uniquement)
– gros travail sur du code mal conçu
– introduction de bugs en parallélisant du code non parallélisable

Intel Threading Building Blocks

- Bibliothèque pour C++
- Structures de contrôles 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 (compilateur + matériel)

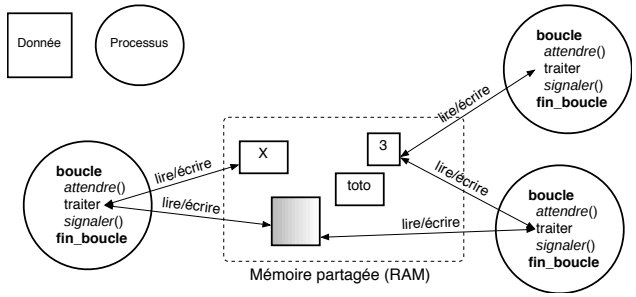
Septième partie

Processus communicants

Contenu de cette partie

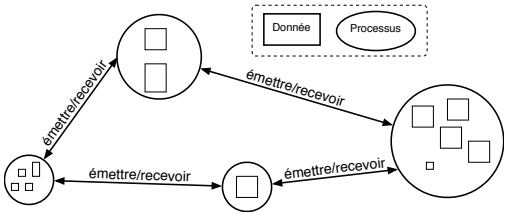
- Modèles de programmation concurrente
- Modèle des processus communicants
- Approche CSP/Go pour la programmation concurrente
  - Goroutine et canaux
  - Communiquer explicitement plutôt que partager implicitement
- Approche Ada pour la programmation concurrente
  - Tâches et rendez vous
  - Démarche 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

Modèles d'interaction : mémoire partagée ♪♪♪



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

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
  - Isolation des données
- Synchronisation implicite : attente de message
- Architectures/modèles cibles
  - systèmes répartis : sites distants, reliés par un réseau
  - moniteurs, CSP/Erlang/Go, tâches Ada

Plan

Processus communicants

Principes

- Communication inter-processus avec des opérations explicites d'envoi / réception de messages
- Synchronisation via ces primitives de communication bloquantes : envoi (bloquant) de messages / réception bloquante de messages

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

Les principes détaillés des échanges et leur utilisation pour développer des applications sont vus dans le module « intergiciels ». On ne s'intéresse ici qu'à la synchronisation.

Quelle synchronisation ?

Réception

Réception bloquante : attendre un message

Émission

- Émission non bloquante ou asynchrone
- Émission bloquante ou synchrone : bloque jusqu'à la réception du message = rendez-vous élémentaire entre l'activité émettrice et l'activité destinataire
- Rendez-vous étendu : bloquant jusqu'à réception + réaction + réponse $\approx$ appel de procédure

- Émission asynchrone $\Rightarrow$ buffers (messages émis non reçus)
- Synchrone $\Rightarrow$ 1 case suffit

Désignation du destinataire et de l'émetteur

Nommage

- Direct : désignation de l'activité émettrice/destinataire

```
SEND message TO processName
RECV message FROM processName
```

- Indirect : désignation d'une boîte à lettres ou d'un canal de communication

```
SEND message TO channel
RECV message FROM channel
```

Multiplicité

1 – 1

Désignation de l'activité : 1 émetteur / 1 récepteur désignés

n – 1

Canal réservé en lecture (consommation) : envoi par n'importe quelle activité ; réception par une seule, propriétaire du canal

n – m

Canal avec envoi par n'importe qui, réception par n'importe qui :

- pas de duplication : un seul destinataire consomme le message
- ou duplication à tous les destinataires (diffusion)

En mode synchrone, la diffusion est complexe et coûteuse à mettre en œuvre (nécessite une synchronisation globale entre tous les récepteurs)

Alternative

Alternative en émission ou en réception = choix parmi un ensemble de communications possibles :

```
RECV msg FROM channel1 OR channel2
(SEND msg1 TO pid1) OR (SEND msg2 TO pid2)
(RECV msg1 FROM channel1) OR (SEND msg2 TO channel2)
```

- Si aucun choix n'est faisable ⇒ attendre
- Si un seul des choix est faisable ⇒ le faire
- Si plusieurs choix sont faisables ⇒ sélection non-déterministe (arbitraire)

Divers

Émission asynchrone ⇒ risque de buffers pleins

- perte de messages ?
- ou l'émission devient bloquante si plein ?

Émission non bloquante → émission bloquante

introduire un acquittement

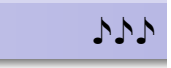
```
   (SEND m TO ch; RECV _ FROM ack)
|| (RECV m FROM ch; SEND _ TO ack)
```

Émission bloquante → émission non bloquante

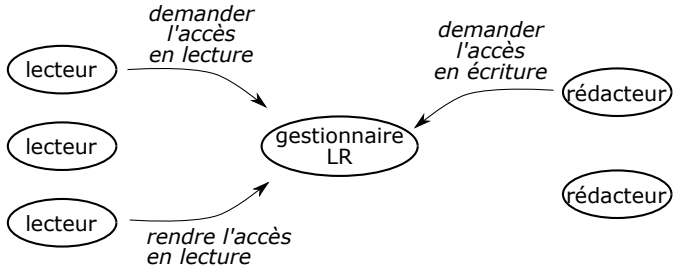
introduire une boîte intermédiaire qui accepte immédiatement tout message et le stocke dans une file.

```
   (SEND m TO ch1)
|| boucle (RECV m FROM ch1; insérer m dans file)
|| boucle (si file non vide alors extraire et SEND TO ch2)
|| (RECV FROM ch2)
```

Architecture



La résolution des problèmes de synchronisation classiques (producteurs/consommateurs...) ne se fait plus en synchronisant directement les activités via des données partagées, mais indirectement via une activité de synchronisation.



Activité arbitre pour un objet partagé

Interactions avec l'objet partagé

Pour chaque opération $\mathcal{O}p$,

- émettre un message de requête vers l'arbitre
- attendre le message de réponse de l'arbitre

(⇒ se synchroniser avec l'arbitre)

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 message correspondant)

Note : en communication synchrone, on peut se passer du message de réponse s'il n'y a pas de contenu à fournir.

Intérêt

- \+ découplage entre les activités clientes : l'interface partagée est celle de l'activité de synchronisation
- \+ réalisation centralisée et répartie
- \+ transfert explicite d'information : traçage
- \+ pas de données partagées ⇒ pas de protection nécessaire
- \+ contrôle fin des interactions
- \+ schéma naturel côté client : question/réponse = appel de fonction
- – multiples recopies (mais optimisations possibles)
- – parallélisation du service : au cas par cas

Plan

Go language

Principes de conception

- Syntaxe légère inspirée du C
- Typage statique fort avec inférence
- Interfaces avec extension et polymorphisme (typage structurel / duck typing à la Smalltalk)
- Ramasse-miettes

Concepts pour la concurrence

- Descendant de CSP (Hoare 1978), cousin d'Erlang
- Goroutine ∼ activité/thread
  - une fonction s'exécutant indépendant (avec sa pile)
  - très léger (plusieurs milliers sans problème)
  - gérée par le noyau Go qui alloue les ressources processeurs
- Canaux pour la communication et la synchronisation

Go – canaux

Canaux

- Création : `make(chan type)` ou `make(chan type, 10)` (synchrone / asynchrone avec capacité)
- Envoi d'une valeur sur le canal `chan` : `chan <- valeur`
- Réception d'une valeur depuis `chan` : `<- chan`
- Canal transmissible en paramètre ou dans un canal : `chan chan int` est un canal qui transporte des canaux (transportant des entiers)

Go – canaux

Alternative en réception et émission

```
select {
 case v1 := <- chan1:
      fmt.Printf("received %v from chan1\n", v1)
 case v2 := <- chan2:
      fmt.Printf("received %v from chan2\n", v2)
 case chan3 <- 42:
      fmt.Printf("sent %v to chan3\n", 42)
 default:
      fmt.Printf("no one ready to communicate\n")
}
```

Exemple élémentaire

```
func boring(msg string, c chan string) {
  for i := 0; ; i++ {
    c <- fmt.Sprintf("%s %d", msg, i)
    time.Sleep(time.Duration(rand.Intn(4)) * time.Second)
  }
}
```

```
func main() {
  c := make(chan string)
  go boring("boring!", c)
  for i := 0; i < 5; i++ {
     fmt.Printf("You say: %q\n", <- c)
  }
  fmt.Println("You're boring; I'm leaving.")
}
```

Moteur de recherche

Objectif : agrégation de la recherche dans plusieurs bases

```
func Web(query string) Result
func Image(query string) Result
func Video(query string) Result
```

Moteur séquentiel

```
func Google(query string) (results []Result) {
     results = append(results, Web(query))
     results = append(results, Image(query))
     results = append(results, Video(query))
    return
}
```

exemple tiré de `https://talks.golang.org/2012/concurrency.slide`

Recherche concurrente

Moteur concurrent

```
func Google(query string) ( results []Result) {
    c := make(chan Result)
    go func() { c <- Web(query) } ()
    go func() { c <- Image(query) } ()
    go func() { c <- Video(query) } ()

    for i := 0; i < 3; i++ {
        result := <- c
        results = append(results, result)
    }
    return
}
```


Le temps sans interruption

Crée un canal sur lequel un message sera envoyé après la durée spécifiée.

time.After

```
func After(d time.Duration) <-chan bool {
    // Returns a receive-only channel
    // A message will be sent on it after the duration
    c := make(chan bool)
    go func() {
        time.Sleep(d)
        c <- true
    }()
  return c
}
```


Recherche concurrente en temps borné

Moteur concurrent avec timeout

```
c := make(chan Result)
go func() { c <- Web(query) } ()
go func() { c <- Image(query) } ()
go func() { c <- Video(query) } ()

timeout := time.After(80 * time.Millisecond)
for i := 0; i < 3; i++ {
    select {
        case result := <-c:
            results = append(results, result)
        case <-timeout:
            fmt.Println("timed out")
            return
    }
}
return
```

Recherche répliquée

Utiliser plusieurs serveurs répliqués et garder la réponse du premier qui répond.

Recherche en parallèle

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result)
    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```

Recherche répliquée

Moteur concurrent répliqué avec timeout

```
c := make(chan Result)
go func() { c <- First(query, Web1, Web2, Web3) } ()
go func() { c <- First(query, Image1, Image2) } ()
go func() { c <- First(query, Video1, Video2) } ()
timeout := time.After(80 * time.Millisecond)
for i := 0; i < 3; i++ {
    select {
        case result := <-c:
            results = append(results, result)
        case <-timeout:
            fmt.Println("timed out")
            return
    }
}
return
```

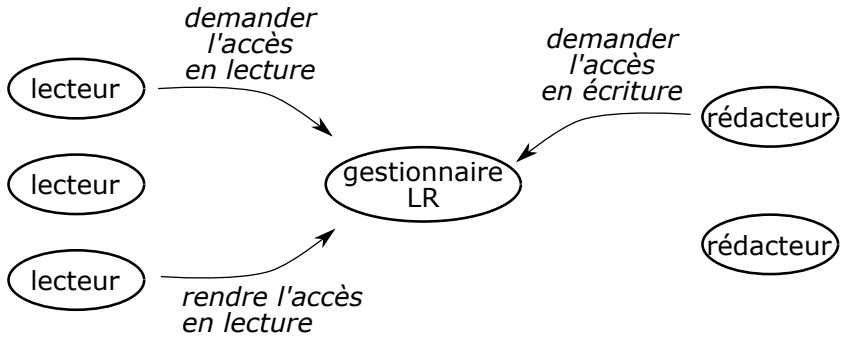

Bilan

- Création ultra-légère de goroutine : penser concurrent
- Pas besoin de variables partagées
  ⇒ Pas de verrous
- Pas besoin de variable condition / sémaphore pour synchroniser
- Pas besoin de callback ou d'interruption

Don't communicate by sharing memory, share memory by communicating.

(la bibliothèque Go contient *aussi* les objets usuels de synchronisation pour travailler en mémoire partagée : verrous, sémaphores, moniteur...)

Lecteurs/rédacteurs



- Un canal pour chaque type de requête : DL, TL, DE, TE
- Émission bloquante ⇒ accepter un message (une requête) uniquement si l'état l'autorise

Lecteurs/rédacteurs

Utilisateur

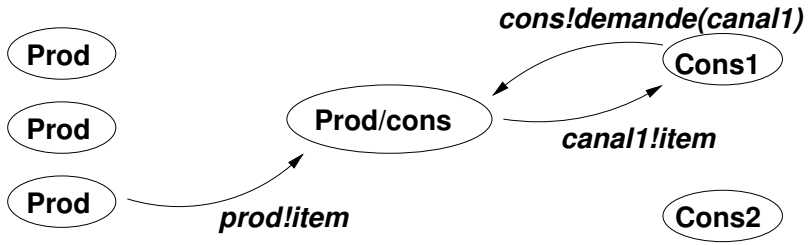
```
func Utilisateur () {
  nothing := struct{}{}
  for {
    DL <- nothing; // demander lecture
    ...
    TL <- nothing; // terminer lecture
    ...
    DE <- nothing; // demander écriture
    ...
    TE <- nothing; // terminer écriture
  }
}
```

Lecteurs/rédacteurs

Goroutine de synchronisation

```
func when(b bool, c chan struct{}) chan struct{} {
    if b { return c } else { return nil }
}

func SynchroLR() {
  nblec := 0;
  ecr := false;
  for {
     select {
        case <- when(nblec == 0 && !ecr, DE):
              ecr := true;
        case <- when(!ecr, DL):
              nblec++;
        case <- TE:
              ecr := false;
        case <- TL:
              nblec--;
     }
}}
```


Producteurs/consommateurs : architecture



- Un canal pour les demandes de dépôt
- Un canal pour les demandes de retrait
- Un canal par activité demandant le retrait (pour la réponse à celle-ci)

(exercice futile : `make(chan T, N)` est déjà un tampon borné = un prod/cons de taille N)

Producteurs/consommateurs

Programme principal

```
func main() {
    prod := make(chan int)      // un canal portant des entiers
    cons := make(chan chan int) // un canal portant des canaux
    go prodcons(prod, cons)
    for i := 1; i < 10; i++ {
        go producteur(prod)
    }
    for i := 1; i < 5; i++ {
        go consommateur(cons)
    }
    time.Sleep (20*time.Second)
    fmt.Println ("DONE.")
}
```


Producteurs/consommateurs

Producteur

```
func producteur(prod chan int) {
  for {
     ...
    item := ...
    prod <- item
  }}
```

Consommateur

```
func consommateur(cons chan chan int) {
  moi := make(chan int)
  for {
     ...
    cons <- moi
    item := <- moi
    // utiliser item
  }}
```

Producteurs/consommateurs

Goroutine de synchronisation

```
func prodcons(prod chan int, cons chan chan int) {
  nbocc := 0;
  queue := make([]int, 0)
  for {
    if nbocc == 0 {
      m := <- prod; nbocc++; queue = append(queue, m)
    } else if nbocc == N {
      c := <- cons; c <- queue[0]; nbocc--; queue = queue[1:]
    } else {
      select {
        case m := <- prod: nbocc++; queue = append(queue, m)
        case c := <- cons:
              c <- queue[0]; nbocc--; queue = queue[1:]
      }
    }
  }
}
```

Plan

Modèle Ada

Intérêt

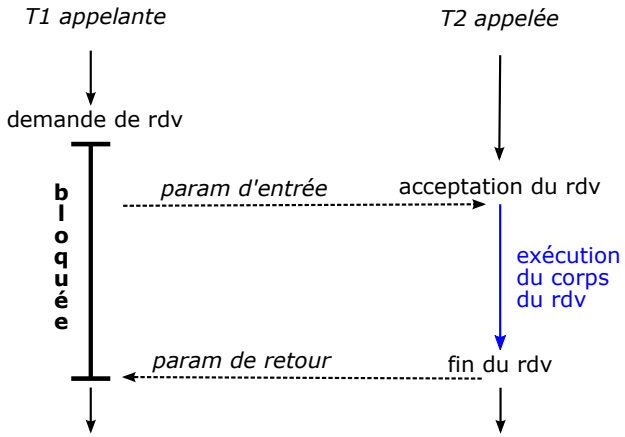
- Modèle adapté à la répartition, contrairement aux sémaphores ou aux moniteurs, intrinsèquement centralisés.
- Similaire au modèle client-serveur.
- Contrôle plus fin du moment où les interactions ont lieu.

Vocabulaire : tâche = activité

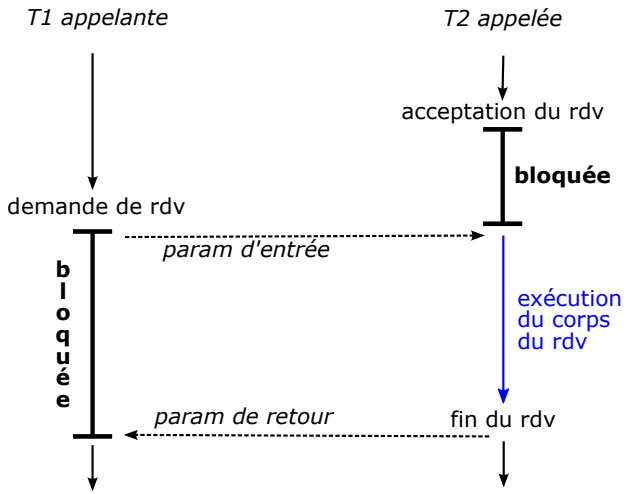
Principe du rendez-vous

- Une tâche possède des points d'entrée de rendez-vous.
- Une tâche peut :
  - demander un rendez-vous avec une autre tâche désignée explicitement ;
  - attendre un rendez-vous sur un (ou plusieurs) point(s) d'entrée.
- Un rendez-vous est dissymétrique : tâche appelante ou cliente vs tâche appelée ou serveur.
- 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.

Rendez-vous – client en premier



Rendez-vous – serveur en premier



Principe du rendez-vous

- 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 indique qu'il est prêt à accepter un rendez-vous et qu'il n'y a pas de demandeur, il se bloque.
- En outre, l'appelant est bloqué pendant l'exécution du corps du rendez-vous.

Important : il est impossible d'accepter/refuser un rendez-vous selon la valeur des paramètres.

Déclaration d'une tâche

Déclaration

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

Exemple

```
task X is
    entry A;
    entry B (msg : in T);
    entry C (x : out T);
    entry D (a : in T1; b : out T2);
end X
```

Appel de rendez-vous

Appel de rendez-vous

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

Syntaxe identique à un appel de procédure, sémantique bloquante.

Exemple

```
X.A;
X.D(x,y);
```

Acceptation d'un rendez-vous

Acceptation

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

Exemple

```
task body X is
begin
  loop
       ...
      accept D (a : in Natural; b : out Natural) do
          if a > 6 then b := a / 4;
          else b := a + 2; end if;
      end D;
  end loop;
end X;
```

Acceptation parmi un ensemble

Alternative gardée

```
select
      when C1 =>
          accept E1 do
              ...
          end E1;
   or
      when C2 =>
          accept E2 do
              ...
          end E2;
   or
      ...
end select;
```

Producteurs/consommateurs

Déclaration du serveur

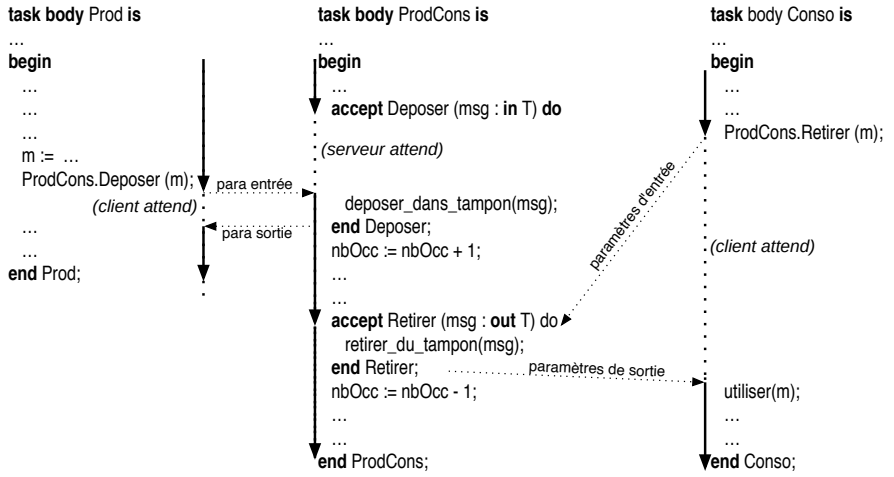
```
task ProdCons is
   entry Deposer (msg: in T);
   entry Retirer (msg: out T);
end ProdCons;
```

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;
      end select;
   end loop;
end ProdCons;
```


Producteurs/consommateurs – un exemple d'exécution



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.
- `rdv'count` (attribut des entrées) donne le nombre de clients en attente sur une entrée donnée.
- 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 ;
  - plusieurs appels à un même service peuvent déclencher des traitements différents ;
  - le serveur peut être bloqué, tandis que des clients attendent.

Allocateur de ressources ♪♪♪

Un système comporte des ressources critiques c'est-à-dire non partageables et non préemptibles, comme les pages mémoire. L'allocateur de ressources est un service qui permet à un processus d'acquérir par une seule action plusieurs ressources. On ne s'intéresse qu'à la synchronisation et on ne s'occupe pas de 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;
```

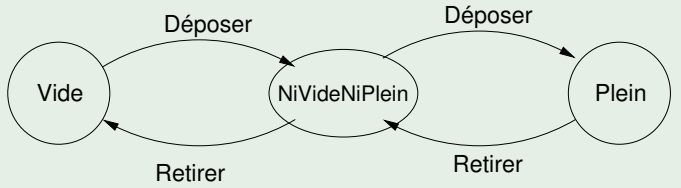
```
task body Allocateur is
   nbDispo : integer := N;
begin
   loop
      select
         accept Demander (nbDemandé : in natural) do
            while nbDemandé > nbDispo loop
               accept Rendre (nbRendu : in natural) do
                  nbDispo := nbDispo + nbRendu;
               end Rendre;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


Méthodologie par machine à états

Construire un automate fini à états :

- identifier les états du système
- un état est caractérisé par les rendez-vous acceptables
- un rendez-vous accepté change (éventuellement) l'état

Producteurs/consommateurs à 2 cases




```
task body ProdCons is
   type EtatT is (Vide, NiVideNiPlein, Plein);
   etat : EtatT := Vide;
begin
   loop
      if etat = Vide then
         select
            accept Deposer (msg : in T) do
               deposer_dans_tampon(msg);
            end Deposer;
            etat := NiVideNiPlein;
         end select;
      elsif etat = NiVideNiPlein then
         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;
```

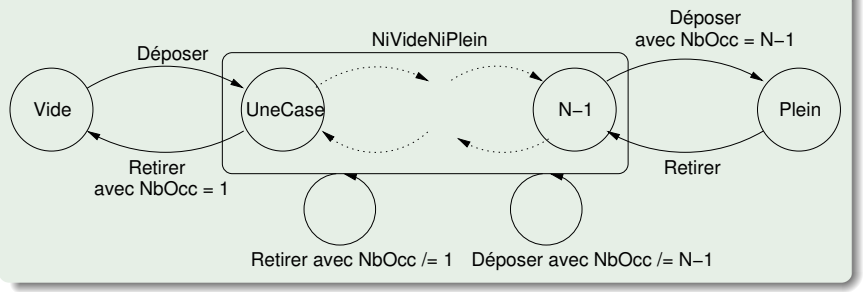


```
      elsif etat = Plein then
         select
            accept Retirer (msg : out T) do
               msg := retirer_du_tampon();
            end Retirer;
            etat := NiVideNiPlein;
         end select;
      end if;
   end loop;
end ProdCons;
```

Automate paramétré

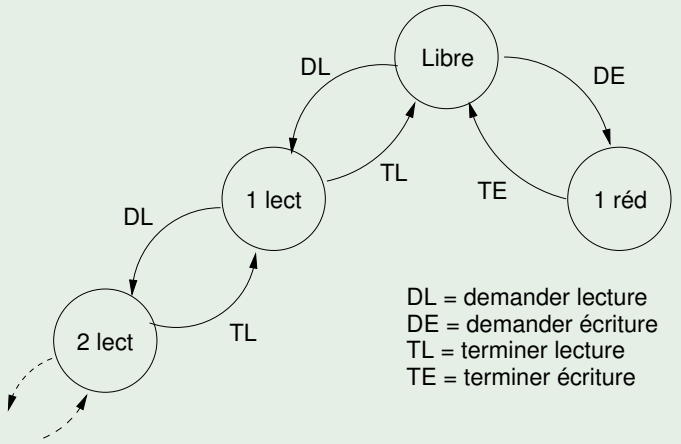
Représenter un *ensemble d'états* comme un unique état *paramétré*.
Les valeurs du paramètre différenciant les états de l'ensemble peuvent être utilisées pour étiqueter les transitions.

Producteurs/consommateurs à N cases



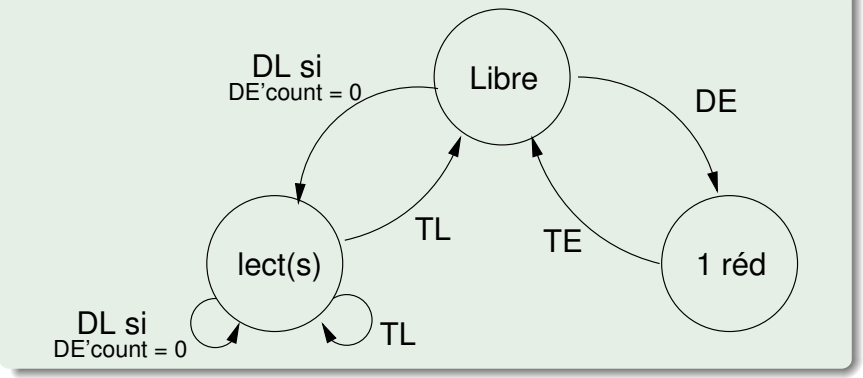
Lecteurs/rédacteurs

Lecteurs/rédacteurs



Lecteurs/rédacteurs priorité rédacteurs

Lecteurs/rédacteurs priorité rédacteurs




```
task body LRprioRed is
   type EtatT is (Libre, Lect, Red);
   etat : EtatT := Libre;
   nblect : Natural := 0;
begin
  loop
    if etat = Libre then
      select
        when DE'count = 0 => accept DL; etat := Lect; nblect := 1;
      or
        accept DE; etat := Red;
      end select;
    elsif etat = Lect then
      select
        when DE'count = 0 => accept DL; nblect := nblect + 1;
      or
        accept TL; nblect := nblect - 1;
        if nblect = 0 then etat := Libre; else etat := Lect; end if;
      end select;
    elsif etat = Red then
      accept TE;
      etat := Libre;
    end if;
  end loop;
end LRprioRed;
```

Dynamicité : activation de tâche

Une tâche peut être activée :

- statiquement : chaque **task** T, déclarée explicitement, est activée au démarrage du programme, avant l'initialisation des modules 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.

Dynamicité :Terminaison

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.

La terminaison est difficile !

Bilan processus communicants ♪♪♪

\+ Pas de partage implicite de la mémoire ($\rightarrow$ isolation)
\+ Transfert explicite d'information ($\rightarrow$ traçage)
\+ Réalisation centralisée et répartie
\+ Contrôle fin des interactions
$\sim$ Méthodologie
– Performance (copies)
– Quelques schémas classiques, faire preuve d'invention ($\rightarrow$ attention aux doigts)

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

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*
  → contrôle explicite de l'attente/la progression des processus
- utilisation d'un service : *concurrence*
  → partage transparent : chaque processus interagit avec l'objet partagé comme s'il était seul à l'utiliser.

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
- transparent : du point de vue du programmeur tout se passe comme si son traitement était seul à s'exécuter
- 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

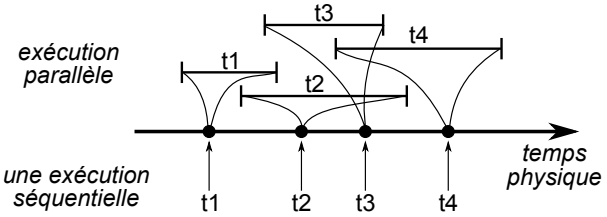
- Systèmes d'information : bases de données → intergiciels
- Mémoire transactionnelle (architectures mutiprocesseurs) (HTM/STM = hardware/software transactional memory)

Service transactionnel : propriétés

Les états intermédiaires (entre l'état initial et l'état final) ne sont pas observables en dehors de la transaction

- si la transaction se termine bien seul son état final est visible.
- si la transaction ne peut se terminer (échec, *abandon*...), elle n'a aucun effet visible.

Autre formulation : le service transactionnel contrôle les effets des transactions afin que « tout se passe comme si » l'exécution de chaque transaction était instantanée, se réduisait à un point dans le temps.



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 $\geq$ 0 et Y.val $\geq$ 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`

Interférences et cohérence : exemple

T1, T2 : traitements s'exécutant sans service transactionnel.

Invariant $x + y = z$
Initialement $x = 500, y = 500, z = 1000$

$T1$		$T2$
1. $a_1 \leftarrow x$	$\alpha.$	$c_2 \leftarrow z$
2. $b_1 \leftarrow y$	$\beta.$	$z \leftarrow c_2 + 200$
3. $x \leftarrow a_1 - 100$	$\gamma.$	$d_2 \leftarrow x$
4. $y \leftarrow b_1 + 100$	$\delta.$	$x \leftarrow d_2 + 200$

OK : $\langle 1 \cdot 2 \cdot \alpha \cdot \beta \cdot 3 \cdot \gamma \cdot 4 \cdot \delta \rangle (x = 600, y = 600, z = 1200 = x + y)$
KO : $\langle 1 \cdot 2 \cdot \alpha \cdot \beta \cdot \gamma \cdot 3 \cdot 4 \cdot \delta \rangle (x = 700, y = 600, z = 1200 \neq x + y)$
Remarque : comme T1 et T2 maintiennent l'invariant,
les exécutions séquentielles (T1 ;T2 et T2 ;T1) le maintiennent aussi.

(x, y, z : données partagées ;
a_1, b_1, c_2, d_2 variables locales privées aux traitements)

Plan

Atomicité (tout ou rien)

Objectif

- Intégrer les résultats des transactions « bien » terminées :
  si une transaction valide (avec succès) tous ses résultats sont définitivement visibles (« tout »)
- Assurer qu'une transaction annulée n'a aucun effet sur les données partagées :
  si une transaction échoue à valider ou bien abandonne, elle n'a aucun effet sur les autres transactions : tout se passe comme si elle ne s'était pas exécutée (« rien »)

Comment abandonner une transaction sans effet ?

Deux stratégies possibles

- *Pessimiste / propagation différée* : mémoire temporaire transférée en mémoire définitive à la validation
- *Optimiste / propagation immédiate (en continu)* : écriture directe avec sauvegarde de l'ancienne valeur ou de l'action inverse

Difficultés

- Tenir compte de la possibilité de pannes en cours
  - d'exécution,
  - ou d'enregistrement des résultats définitifs,
  - ou d'annulation.
- Effet domino : T' observe une écriture de T puis T abandonne $\Rightarrow$ T' doit être abandonnée

Mise en œuvre de l'atomicité

Opérations de base

- *défaire* : revenir à l'état initial d'une transaction annulée
- *refaire* : 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

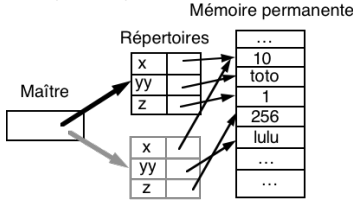
- écrire → préécriture, dans l'espace de travail
- valider → recopier l'espace de travail en mémoire stable (*liste d'intentions*), puis copier celle-ci en mémoire permanente
  → protection contre les pannes en cours de validation
- défaire → libérer l'espace de travail
- refaire → reprendre la recopie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

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.

Complément : nuances autour de la sérialisabilité

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

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.

Remarque

La notion de conflit n'est pas spécifique aux opérations *lire/écrire*.
$\rightarrow$ généralisation : définir un tableau de commutativité entre actions

Exemple

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

Notion de conflit

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

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

$\rightarrow$ opérations en *conflit* :

opérations non commutatives exécutées sur un même objet

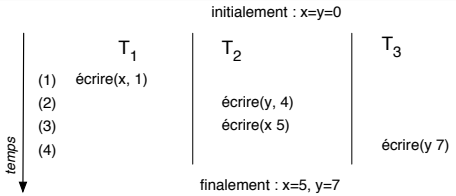
Exemple principal : opérations *lire(x)* et *écrire(x,v)*

- 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') ;

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)

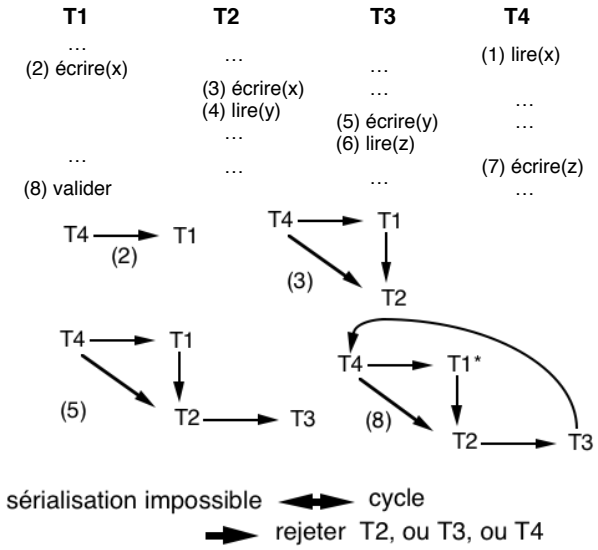
Définitions

- Relation de dépendance $\rightarrow$: $T_1 \rightarrow T_2$ 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 [Papadimitriou]

Exécution sérialisable $\Leftrightarrow$ 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/évaluer 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)

Plan

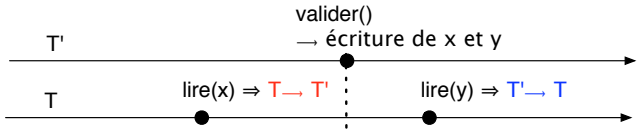
Contrôle de concurrence par certification : principe

- **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 → écritures de T pas encore effectuées

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



Plan

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
```

rejet coûteux ⇒ faible taux de conflit

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

```
procédure écrire(T,O,v)
si T.E ≥ O.lect ∧ T.E ≥ O.réd
alors  /* écriture de O possible */
   écriture effective
   O.red ← T.E
sinon
   abandon de T
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 peut être 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) s : 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 exécutées dans le même ordre
→ pas de dépendances d'orientation opposée → pas de cycle

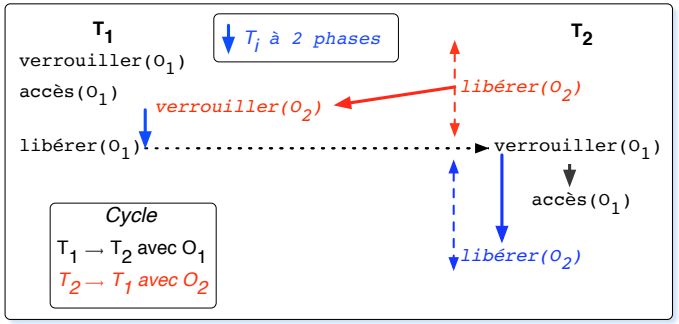
Illustration (sur un contre-exemple)

Invariant $x = y$ (*initialement* $x = y = 4$)

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

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

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

Plan

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
  → *cohérence faible*

[Exemple de cohérence faible : niveaux d'isolation SQL (1/3)]

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)

[Niveaux d'isolation SQL (2/3)]

Pertes de mises à jour

Écritures écrasées par d'autres écritures.

(1) a := lire(x) ;	‖	(a) b := lire(x) ;
(2) écrire(x,a+10) ;	‖	(b) écrire(x, b+20) ;

Lectures sales

Écritures abandonnées mais observées

(1) écrire(x,100) ;	‖	(a) b := lire(x) ;
(2) abandon ;	‖	

[Niveaux d'isolation SQL (3/3)]

Lectures non répétables

Donnée qui change de valeur pendant la transaction

(1) a := lire(x) ;	‖	(a) écrire(x,100) ;
(2) b := lire(x) ;	‖	

Lectures fantômes

Donnée agrégée qui change de contenu

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

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
- Importance de la gestion des transactions afin de limiter/prévenir les conflits (*gestion de contention*) :
  - quelles transactions bloquer/rejeter en cas de conflit ?
  - quelles transactions débloquer/relancer, et quand ?

Optimisme (≈ détection-guérison) et pessimisme (≈ prévention)

Approches optimistes ou pessimistes envisageables pour 2 fonctions très distinctes

Propagation des valeurs écrites

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

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

Contrôle de concurrence

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

→ Cohérence et isolation, comme si chaque transaction était seule

Ces deux politiques se combinent ± bien.

Plan

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

Si la condition nécessaire à la progression n'est pas vérifiée

- (Attendre que les variables accédées aient changé de valeur)
- (Annuler puis) Relancer (automatiquement) la transaction

→ opération retry

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

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

Plan

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(Tk);
return "ok";
```

- $abort(T_k)$

```
// restaurer les valeurs ecrites
foreach m ∈ Tk.ecrits do Mem[m].write(Tk.SvMem[m]);
unlock_all(Tk);
return "ok";
```

- $unlock\_all(T)$

```
// liberer tous les verrous obtenus par T
foreach m ∈ T.lus ∪ T.ecrits do L[m].unlock();
T.lus := ∅ ;
T.ecrits := ∅ ;
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)

Implantation hybride

Coopération STM/HTM

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

Implantation STM sur HTM

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

Implantation STM avec assistance matérielle

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

Difficultés du modèle (1/3)

Propagation directe ⇒ effets de bord

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

```
                      init x=y
atomic {                  | atomic {
     if (nonnul)          |      x ← NULL;
         *x ← 3;          |      nonnul ← false;
}                         | }
```

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

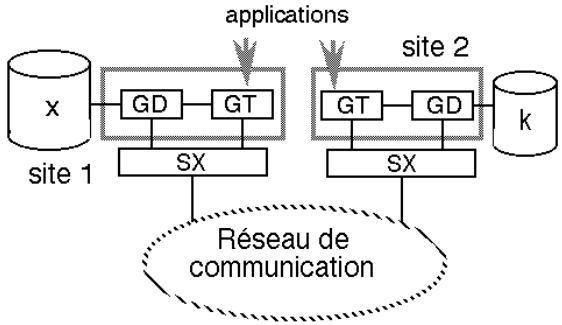
Une transaction annulée doit être sans effet : comment faire s'il y a des effets de bords (p.e. entrées/sorties), avec un contrôle de concurrence optimiste ?

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 avec effet de bord, 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

Limitation des verrous ♪♪♪

Limites des verrous (et plus généralement de la synchronisation par blocage/attente) :

- Interblocage : ensemble de processus se bloquant mutuellement
- Inversion de priorité : un processus de faible priorité bloque un processus plus prioritaire
- Convoi : une ensemble d'actions avance à la vitesse de la plus lente
- Interruption : quelles actions dans un gestionnaire de signal ?
- Arrêt involontaire d'un processus
- Tuer un processus ?
- Granularité des verrous → performance

Objectifs de la synchronisation non bloquante ♪♪♪

Problème

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

- Résistance à l'arrêt (crash) d'une activité : une activité donnée n'est jamais empêchée de progresser, quel que soit le comportement des autres activités
- Vitesse de progression indépendante de celle des autres activités
- Passage à l'échelle
- Surcoût négligeable de synchronisation en absence de conflit (notion de *fast path*)
- Compatible avec la programmation événementielle (un gestionnaire d'interruption ne doit pas être bloqué par la synchronisation)

Synchronisation non bloquante

Non-blocking synchronization

Obstruction-free Si à tout point, une activité en isolation parvient à terminer en temps fini (en un nombre fini de pas).

Lock-free Synchronisation et protection garantissant la *progression du système* même si une activité s'arrête arbitrairement. Peut utiliser de l'attente active mais (par exemple) pas de verrous.
Absence d'interblocage et d'inversion de priorité mais risque de famine individuelle (vivacité faible).

Wait-free Une sous-classe de lock-free où *toute activité* est certaine de compléter son action en temps fini, indépendamment du comportement des autres activités (arrêtées ou agressivement interférantes).
Absence de famine individuelle (vivacité forte).

Mécanismes matériels

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 : test-and-set)

Principes généraux

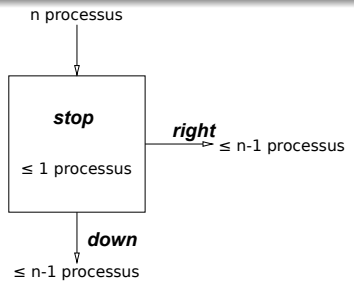
Principes

- Chaque activité 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, une activité peut exécuter des opérations pour le compte d'une autre activité (p.e. finir la mise à jour de l'objet partagé)

Plan

Splitter

Moir, Anderson 1995



- x (indéterminé) activités appellent concurremment (ou pas) le splitter
- au plus une activité termine avec *stop*
- si $x = 1$, l'activité termine avec *stop*
- au plus $(x - 1)$ activités terminent avec *right*
- au plus $(x - 1)$ activités terminent avec *down*

Splitter

Registres

- Lectures et écritures atomiques
- Pas d'interférence due aux caches en multiprocesseur

Implantation non bloquante

Deux registres partagés : X (init $\forall$) et Y (init faux)
Chaque activité a un identifiant unique id_i et un résultat dir_i.

```
function direction (id_i)
  X := id_i;
  if Y then dir_i := right;
  else Y := true;
    if (X = id_i) then dir_i := stop;
    else dir_i := down;
    end if
  end if
  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 (une seule activité exécute *direction()*)

au plus $x - 1$ right les activités obtenant right trouvent Y, qui a nécessairement été positionné par une activité obtenant down ou stop

au plus $x - 1$ down soit p_i la dernière activité ayant écrit X. Si p_i trouve Y, elle obtiendra right. Sinon son test $X = id_i$ lui fera obtenir stop.

au plus 1 stop soit p_i la *première* activité trouvant $X = id_i$. Alors aucune activité n'a modifié X depuis que p_i l'a fait. Donc toutes les activités suivantes trouveront Y et obtiendront right (car p_i a positionné Y), et les activités en cours qui n'ont pas trouvé Y ont vu leur écriture de X écrasée par p_i (puisqu'elle n'a pas changé jusqu'au test par p_i). Elles ne pourront donc trouver X égal à leur identifiant et obtiendront donc down.

Renommage

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

Solution à base de verrous

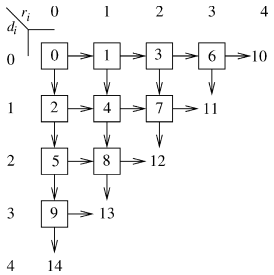
- Distributeur de numéro accédé en exclusion mutuelle
- $M = n$
- Complexité temporelle : O(1) pour un numéro, O(n) pour tous
- Une activité lente ralentit les autres

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 ♪♪♪

Étiquettes uniques : un splitter renvoie stop à une activité au plus



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

Toute activité obtient une étiquette :

- stop si $x = 1$,
- un splitter ne peut orienter toutes les activités dans la même direction,
- les bords de la grille sont à distance $n - 1$ de l'origine.

Renommage non bloquant

get_name(id_i)

$d_i \leftarrow 0$; $r_i \leftarrow 0$; $term_i \leftarrow false$;
while $(\neg term_i)$ **do**

$X[d_i, r_i] \leftarrow id_i$;
if $Y[d_i, r_i]$ **then** $r_i \leftarrow r_i + 1$; % *right*
else $Y[d_i, r_i] \leftarrow true$;
if $(X[d_i, r_i] = id_i)$ **then** $term_i \leftarrow true$; % *stop*
else $d_i \leftarrow d_i + 1$; % *down*
endif
endif

endwhile
return $\frac{1}{2}(r_i + d_i)(r_i + d_i + 1) + d_i$
% *le nom en position* d_i, r_i *de la grille*

Pile chaînée basique ♪♪♪

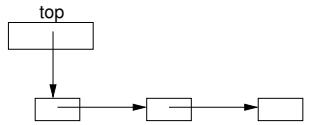
Objet avec opérations push et pop

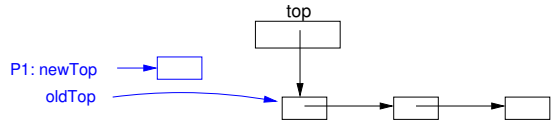
```
class Stack<T> {
  class Node<T> { Node<T> next; T item; }

  Node<T> top;

 public void push(T item) {
   Node<T> newTop
           = new Node<>(item);
   Node<T> oldTop = top;
   newTop.next = oldTop;
   top = newTop;
 }
}
```

```
 public T pop() {
   Node<T> oldTop = top;
   if (oldTop == null)
       return null;
   top = oldTop.next;
   return oldTop.item;
 }
```

Non résistant à une utilisation concurrente par plusieurs activités

Pile chaînée basique : conflit push/push





Synchronisation classique

Conflit push/push, pop/pop, push/pop ⇒ exclusion mutuelle

```
public void push(T item) {
    verrou.lock();
    Node<T> newTop
          = new Node<>(item);
    Node<T> oldTop = top;
    newTop.next = oldTop;
    top = newTop;
    verrou.unlock();
}
```

```
public T pop() {
    verrou.lock();
    try {
        Node<T> oldTop = top;
        if (oldTop == null)
            return null;
        top = oldTop.next;
        return oldTop.item;
    } finally {
        verrou.unlock();
    }
}
```

- Bloquant définitivement si une activité s'arrête en plein milieu
- Toutes les activités sont ralenties par un unique lent

Pile chaînée non bloquante

Principe du `push`

1. Préparer une nouvelle cellule (valeur à empiler)
2. Chaîner cette cellule avec le sommet actuel
3. Si le sommet n'a pas changé, le mettre à jour avec la nouvelle cellule. *cette action doit être atomique !*
4. Sinon, recommencer à l'étape 2

Principe du `pop`

1. Récupérer la cellule au sommet
2. Récupérer la cellule suivante celle au sommet
3. Si le sommet n'a pas changé, le mettre à jour avec celle-ci. *cette action doit être atomique !*
4. Sinon, recommencer à l'étape 1
5. Retourner la valeur dans l'ancien sommet

Registres et Compare-and-set

`java.util.concurrent.atomic.AtomicReference`

- Lectures et écritures atomiques (registres atomiques), sans interférence due aux caches en multiprocesseur
- Une instruction atomique évoluée : `compareAndSet`

```
public class AtomicReference<V> { /* simplified */
  private volatile V value; /* la valeur contenue dans le registre */
  public V get() { return value; }
  public boolean compareAndSet(V expect, V update) {
      atomically {
          if (value == expect) { value = update; return true; }
          else { return false; }
      }
  }
}
```


Push/pop lock free

```
class Stack<T> {
    class Node<T> { Node<T> next; T item; }
    AtomicReference<Node<T>> top = new AtomicReference<>();

    public void push(T item) {
        Node<T> oldTop, newTop = new Node<>();
        newTop.item = item;
        do {
            oldTop = top.get();
            newTop.next = oldTop;
        } while (! top.compareAndSet(oldTop, newTop));
    }

    public T pop() {
        Node<T> oldTop, newTop;
        do {
            oldTop = top.get();
            if (oldTop == null)
              return null;
            newTop = oldTop.next;
        } while (! top.compareAndSet(oldTop, newTop));
        return oldTop.item;
    }
}
```

File chaînée basique ♪♪♪

```
class Node<T> { Node<T> next; T item; }

class File<T> {
  Node<T> head, queue;
  File() { // noeud bidon en tête
      head = queue = new Node<T>();
  }

void enqueue (T item) {
    Node<T> n = new Node<T>();
    n.item = item;
    queue.next = n;
    queue = n;
}
```

```
T dequeue () {
    T res = null;
    if (head != queue) {
        head = head.next;
        res = head.item;
    }
    return res;
}
```

Non résistant à une utilisation concurrente par plusieurs activités

Synchronisation classique

Conflit enfiler/enfiler, retirer/retirer, enfiler/retirer
⇒ tout en exclusion mutuelle

```
void enqueue (T item) {
    Node<T> n = new Node<T>();
    n.item = item;
    verrou.lock();
    queue.next = n;
    queue = n;
    verrou.unlock();
}
```

```
T dequeue () {
    T res = null;
    verrou.lock();
    if (head != queue) {
        head = head.next;
        res = head.item;
    }
    verrou.unlock();
    return res;
}
```

- Bloquant définitivement si une activité s'arrête en plein milieu
- Toutes les activités sont ralenties par un unique lent
- Compétition systématique enfiler/défiler

File non bloquante

- Toute activité doit s'attendre à trouver une opération *enqueue* à moitié finie, et aider à la finir
- Invariant : l'attribut queue est toujours soit le dernier nœud, soit l'avant-dernier nœud.
- Présent dans la bibliothèque java (`java.util.concurrent.ConcurrentLinkedQueue`)

Par lisibilité, on utilise CAS (compareAndSet) défini ainsi :

```
boolean CAS(*add, old, new) {
    atomically {
        if (*add == old ) { *add = new; return true; }
        else { return false ; }
    }
}
```


Enfiler non bloquant

enqueue non bloquant

```
Node<T> n = new Node<T>;
n.item = item;
do {
   Node<T> lqueue = queue;
   Node<T> lnext = lqueue.next;
   if (lqueue == queue) {      // lqueue et lnext cohérents ?
      if (lnext == null) {     // queue vraiment dernier ?
         if CAS(lqueue.next, lnext, n) // essai lien nouveau noeud
            break;             // succès !
      } else {                 // queue n'était pas le dernier noeud
         CAS(queue, lqueue, lnext); // essai mise à jour queue
      }
   }
} while (1);
CAS(queue, lqueue, n); // insertion réussie, essai m. à j. queue
```

dequeue non bloquant

```
do {
   Node<T> lhead = head;
   Node<T> lqueue = queue;
   Node<T> lnext = lhead.next;
   if (lhead == head) { // lqueue, lhead, lnext cohérents ?
      if (lhead == lqueue) { // file vide ou queue à la traîne ?
         if (lnext == null)
            return null;        // file vide
         CAS(queue, lqueue, lnext); // essai mise à jour queue
      } else {                  // file non vide, prenons la tête
         res = lnext.item;
         if CAS(head, lhead, lnext)  // essai mise à jour tête
            break;                   // succès !
      }
   }
} while (1); // sinon (queue ou tête à la traîne) on recommence
return res;
```

Problème A-B-A

- L'algorithme précédent n'est correct qu'en absence de recyclage des cellules libérées par `dequeue`
- Problème A-B-A :
  1. A_1 lit x et obtient a
  2. A_2 change x en b et libère a
  3. A_3 demande un objet libre et obtient a
  4. A_3 change x en a
  5. A_1 effectue CAS(x,a,...), qui réussit et lui laisse croire que x n'a pas changé depuis sa lecture

Solutions au problème A-B-A

- Compteur de générations, incrémenté à chaque modification
  $\langle a, \text{gen } i\rangle \neq \langle a, \text{gen } i+1\rangle$
  Nécessite un CAS2(x,a,gen,i,...)
  (`java.util.concurrent.atomic.AtomicStampedReference`)
- Instructions load-link / store-conditional (LL/SC) :
  - Load-link renvoie la valeur courante d'une case mémoire
  - Store-conditional écrit une nouvelle valeur à condition que la case mémoire n'a pas été écrite depuis le dernier load-link.
  - (les implantations matérielles imposent souvent des raisons supplémentaires d'échec de SC : imbrication de LL, écriture sur la ligne de cache voire écriture quelconque...)
- Ramasse-miette découplé : retarder la réutilisation d'une cellule (*Hazard pointers*). L'allocation/libération devient alors le facteur limitant de l'algorithme.

Plan

Conclusion

+ performant, même avec beaucoup d'activités
+ résistant à l'arrêt temporaire ou définitif d'une activité
– structure de données ad-hoc
– implantation fragile, peu réutilisable, pas extensible
– implantation très complexe, à réserver aux experts
– implantation liée à une architecture matérielle
– nécessité de prouver la correction
+ bibliothèques spécialisées
  `java.util.concurrent.ConcurrentLinkedQueue`
  `j.u.concurrent.atomic.AtomicReference.compareAndSet`
  `j.u.concurrent.atomic.AtomicInteger`