Systèmes concurrents

2SN - L,A

ENSEEIHT
Département Sciences du Numérique

12 septembre 2023

UE Systèmes concurrents et communicants

Ceci concerne seulement les parcours L et A.

Trois 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
- Examen Intergiciels : écrit
- Projet commun
  - présentation mi-octobre, rendu final mi janvier
  - travail en groupe

Matière : systèmes concurrents – organisation

Composition

- Cours (36%) : définitions, principes, modèles
- TD (28%) : conception et méthodologie
- TP (36%) : implémentation des schémas et principes

Evaluation

- écrit

Pages de l'enseignement : `http://moodle-n7.inp-toulouse.fr`
Contact : `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, exclusion mutuelle, interblocage (C1)
2. Synchronisation à base de moniteur (C2, TD1-2, TP1-2)
3. API Java, Posix Threads (C3, TP3)
4. Processus communicants (C4, TD3, TP4)

Première partie

Introduction

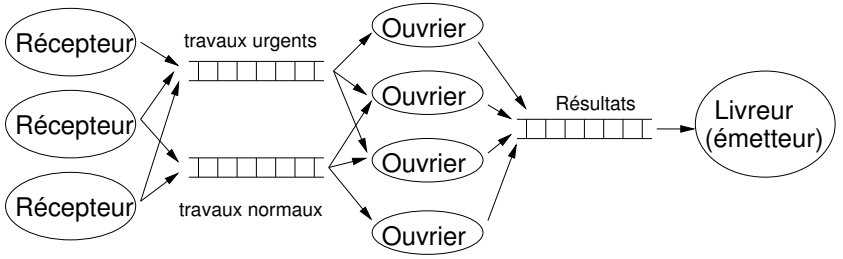
Contenu de cette partie

- Nature et particularités des programmes concurrents
- 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

Exemple de problème

Concevoir une application concurrente qui reçoit des demandes de travaux, les régule, et fournit leur résultat



- coopération : les activités ≪ se connaissent ≫
- compétition : les activités ≪ s'ignorent ≫
- vitesse d'exécution arbitraire

Intérêt des systèmes concurrents

- 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 multiprocesseurs, des stations de travail/ordinateurs en réseau, services répartis

Les architectures multiprocesseurs sont (pour l'instant) le principal moyen d'accroître la puissance de calcul

Différence avec la programmation séquentielle

- Activités ± simultanées ⇒ explosion de l'espace d'états

```
      P1                  ||      P2
for i := 1 to 10          ||  for j := 1 to 10
   print(i)               ||     print(j)
```

  - P1 seul → 10 états ☺
  - P1 ‖ P2 → 10 x 10 = 100 états ☹
  - P1 ; P2 → 1 exécution ☺
  - P1 ‖ P2 → 184756 exécutions ☹
- Interdépendance des activités
  - logique : production/utilisation de résultats intermédiaires
  - chronologique : disponibilité des résultats

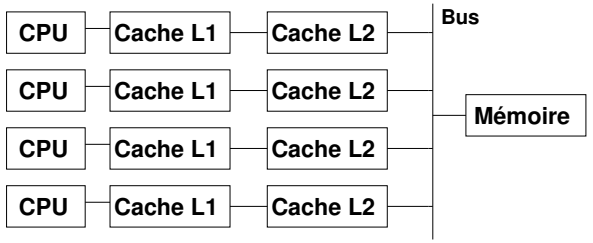
⇒non déterminisme

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

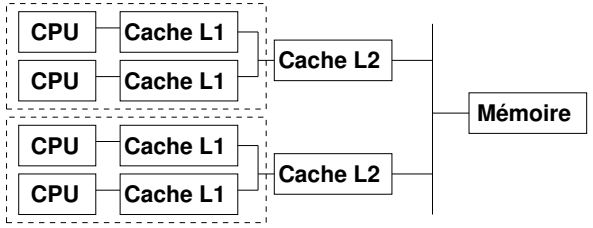
Plan

Architecture multiprocesseur

Multiprocesseur ≪ à l'ancienne ≫ :

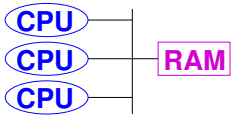


Multiprocesseur multicœur :

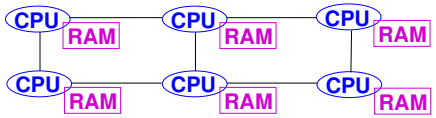


Architecture multiprocesseur

SMP Symmetric multiprocessor : une mémoire + un ensemble de processeurs



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



CC-NUMA Cache-Coherent Non-Uniform Memory Access

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..., sont-elles vues dans cet ordre ? Et les lectures indépendantes d'une écriture ?

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) $r1 \leftarrow y$	(b) $r2 \leftarrow x$

Un résultat $r1 = 0 \wedge r2 = 0$ est possible en cohérence PRAM et slow, impossible en cohérence séquentielle.

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

Processeur P1	Processeur P2
(1) $x \leftarrow 1$	(a) $r1 \leftarrow y$
(2) $y \leftarrow 1$	(b) $r2 \leftarrow x$

Un résultat $r1 = 1 \wedge r2 = 0$ est possible en cohérence slow ou PSO (partial store order – réordonnancement des écritures)

Plan

Activité

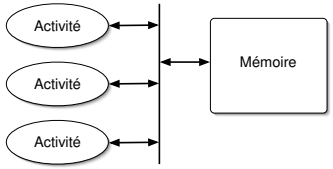
Activité/processus/tâches/threads/processus légers/...

- exécution d'un programme séquentiel
- entité logicielle
- exécutable par un processeur
- interruptible et commutable

Interaction par mémoire partagée

Système centralisé multitâche

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



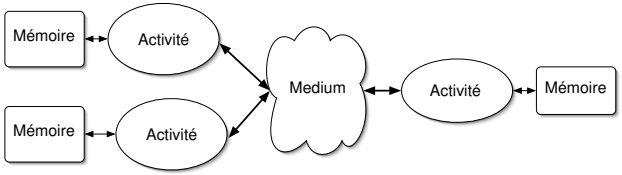
Exemples

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

Interaction par échange de messages

Activités communiquant par messages

- communication explicite par transfert de données (messages)
- désignation nécessaire du destinataire (ou d'un canal de communication)
- coordination implicite, découlant de la communication



Exemples

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

Modèle : l'entrelacement

Raisonner sur tous les cas parallèles est trop complexe
$\Rightarrow$ on raisonne sur des exécutions séquentielles obtenues par entrelacement des instructions des différentes activités.

Deux activités $P = p_1; p_2$ et $Q = q_1; q_2$. L'exécution concurrente $P \| Q$ est vue comme (équivalente à) l'une des exécutions :
$p_1; p_2; q_1; q_2$ ou $p_1; q_1; p_2; q_2$ ou $p_1; q_1; q_2; p_2$ ou $q_1; p_1; p_2; q_2$ ou $q_1; p_1; q_2; p_2$ ou $q_1; q_2; p_1; p_2$

Nombre d'entrelacements : $\frac{(p+q)!}{p!\ q!}$

Attention

- C'est un modèle simplificateur (vraie concurrence, granularité de l'entrelacement, cohérence mémoire...)
- Il peut ne pas exister de code séquentiel équivalent au code parallèle.

Avantages/inconvénients

+ utilisation d'un système multiprocesseur.
+ utilisation de la concurrence naturelle d'un programme.
+ modèle de programmation naturel, en explicitant la synchronisation nécessaire.
– 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 : debuggage souvent délicat (pas de flot séquentiel à suivre) ; effet d'interférence entre des activités, interblocage...

Deuxième partie

L'exclusion mutuelle

Contenu de cette partie

- Difficultés résultant d'accès concurrents à un objet partagé
- Protocoles d'exclusion mutuelle

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.

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. ≫

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 !
```

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 instant, au plus une activité est en cours d'exécution d'une section critique

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

- (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 instant, si p est vrai, alors q sera vrai ultérieurement)

Plan

Instruction matérielle TestAndSet

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
```

Algorithme (Protocole d'exclusion mutuelle)

```
occupé : shared boolean := false;

tant que TestAndSet(occupé) faire nop;
     section critique
occupé ← false;
```

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"

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

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

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.

En particulier les verrous.

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

Définition de l'interblocage

Interblocage

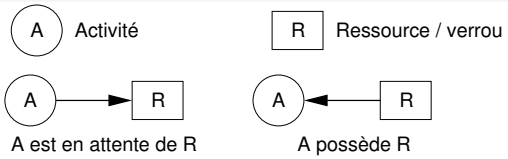
Un ensemble d'activités est en interblocage si et seulement si toute activité de l'ensemble est en attente d'une ressource qui ne peut être libérée que par une autre activité de cet ensemble.

Pour l'ensemble d'activités interbloquées :
Interblocage $\equiv$ négation de la progression

L'interblocage est un état stable.

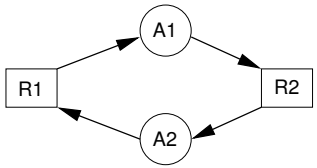
Plan

Notation : graphe d'allocation



Condition nécessaire et suffisante à l'interblocage

Attente circulaire (cycle dans le graphe d'allocation)



Solutions

Prévention : empêcher la formation de cycles dans le graphe

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

Plan

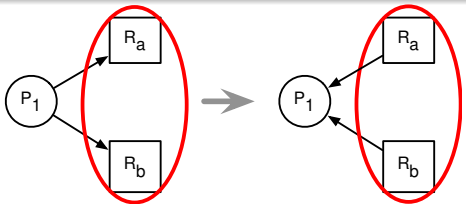
Allocation globale

Éviter les demandes fractionnées

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

→ une seule demande pour chaque activité

- demande satisfaite → arcs entrants uniquement
- demande non satisfaite → arcs sortants (attente) uniquement



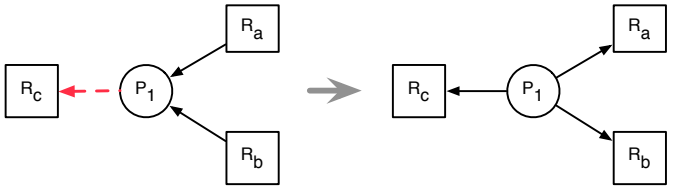
- Revient à remplacer plusieurs verrous par un unique verrou
- Suppose la connaissance a priori des ressources / verrous utilisés
- Sur-allocation, risque de famine, réduction du parallélisme

Non conservation des ressources allouées

Permettre la réquisition des ressources allouées

Inverser les arcs entrants d'une activité si création d'arcs sortants.
Une activité demandeuse doit :

- libérer les ressources qu'elle a obtenues
- réobtenir les ressources libérées, avant de pouvoir poursuivre
  - risque de famine
  - programmation complexe

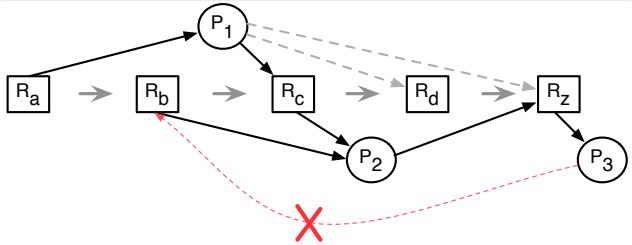


(optimisation : restitution paresseuse des ressources : libération que si la demande est bloquante)

Classes ordonnées

Fixer un ordre global sur les demandes : classes ordonnées

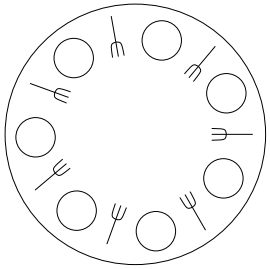
- Un ordre est défini sur les ressources
- Toute activité doit demander les ressources en respectant cet ordre



→ pour chaque activité, 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

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

Détection - guérison

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

- Fixer des critères de choix de l'activité victime (priorités...)
- Annulation du travail effectué par la(les) activité(s) victime(s)

– coûteux (détection + choix + travail perdu + restauration)
– pas toujours acceptable (systèmes interactifs ou embarqués)
– nécessité de points de reprise pour retour arrière
+ allocation simplifiée et plus de parallélisme qu'avec la prévention

Conclusion

Exclusion mutuelle

- Nécessité de bloc atomique "lire-et-écrire" ou "écrire plusieurs variables"
- Solution classique : verrous d'exclusion mutuelle

interblocage

Usuellement : inconvénient occasionnel

- → laissé à la charge de l'utilisateur / du programmeur
- utilisation de méthodes de prévention simples (p.e. classes ordonnées)
- ou détection empirique (délai de garde) et guérison par choix manuel des victimes

Quatrième 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)
- Exemples avancés

Plan

Moniteur de Hoare, Brinch-Hansen (1973)

Idée de base

La synchronisation résulte du besoin de partager convenablement un objet entre plusieurs activités concurrentes

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

Définition

Un moniteur = un module exportant des opérations (et éventuellement des constantes et des types).

- Contrainte :
  exécution des opérations 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 activités utilisant le moniteur qui l'activent, en invoquant ses opérations.

Synchronisation : type *condition*

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

Définition

Variables de type condition définies dans le moniteur.
Opérations possibles sur une variable de type condition C :

- C.wait() (C.attendre()) : bloque l'activité appelante en libérant l'accès exclusif au moniteur.
- C.signal() (C.signaler()) : si des activités sont bloquées sur C, en réveille une ; sinon, nop.

- Une file d'attente est associée à *chaque* variable condition
- condition ≈ événement
  → condition ≠ prédicat logique
- Autre opération sur les conditions :
  C.empty() : vrai si aucune activité n'est bloquée sur C.

Exemple : travail délégué

Schéma client/serveur asynchrone : 1 client + 1 ouvrier

Les activités

Client	Ouvrier
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

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

```
déposer_travail(in t)
  {pas d'attente}
  req ← t
  Dépôt.signal()
```

```
prendre_travail(out t)
  si req = null alors
    Dépôt.wait()
  finsi
  t ← req
  req ← null
  {RAS}
```

```
lire_résultat(out r)
  si rés = null alors
    Dispo.wait()
  finsi
  r ← rés
  rés ← null
  {RAS}
```

```
rendre_résultat(in y)
  {pas d'attente}
  rés ← y
  Dispo.signal()
```


Transfert du contrôle exclusif

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

Priorité au signalé

Lors du réveil par signal(),

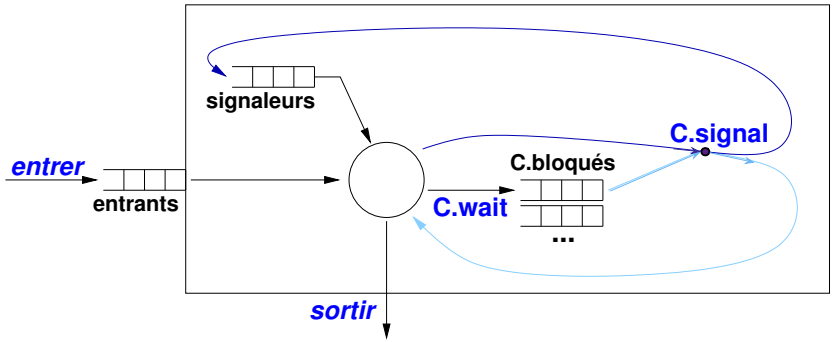
- l'accès exclusif est transféré à l'activité réveillée (signalée) ;
- l'activité signaleuse est mise en attente de pouvoir ré-acquérir l'accès exclusif

Priorité au signaleur

Lors du réveil par signal(),

- l'accès exclusif est conservé par l'activité réveilleuse ;
- l'activité réveillée (signalée) est mise en attente de pouvoir acquérir l'accès exclusif
  - soit dans une file globale spécifique
  - soit avec les activités entrantes

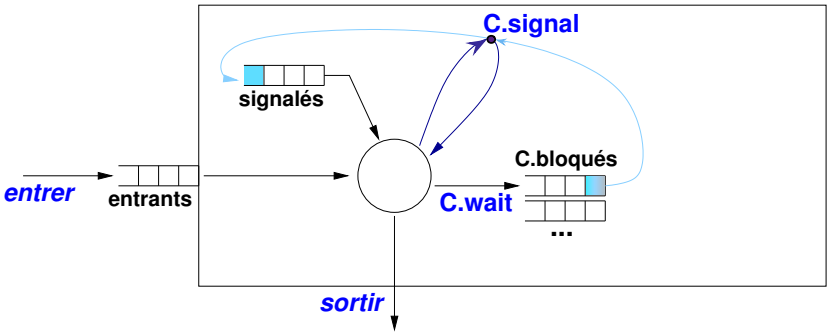
Priorité au signalé



C.*signal*()

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

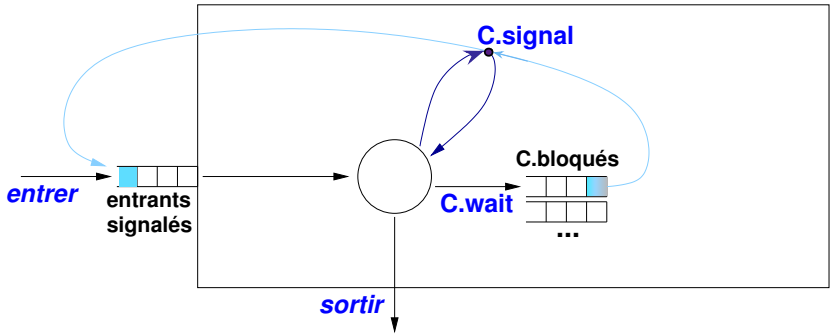
Priorité au signaleur avec file spécifique des signalés



C.*signal*()

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

Signaleur vs signalé : 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.wait()`.
- 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.signal()`, 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 l'activité réveillée 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 l'accès au moniteur ultérieurement, éventuellement après que d'autres activités ont eu accès au moniteur
  - Implantation plus simple et plus performante
  - Au réveil, le signalé doit retester la condition de déblocage

  → Possibilité de famine, écriture et raisonnements plus lourds

Simplifier l'expression de la synchronisation ?

Idée (d'origine)

Attente sur des prédicats, plutôt que sur des événements (= variables de type condition) → opération unique : *wait(B)*, B expression booléenne

Exemple : travail délégué avec *wait*(prédicat)

```
variables d'état : req, rés
-- requête/résultat en attente (null si aucun(e))
```

	`prendre_travail(out t)`
	`  wait(req ≠ null)`
	`  t ← req`
	`  req ← null`

`déposer_travail(in t)` `  req ← t`	

`lire_résultat(out r)`
`  wait(rés ≠ null)`
`  r ← rés`
`  rés ← null`
	`rendre_résultat(in y)`
	`  rés ← y`

Pourquoi *wait*(prédicat) n'est-il 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

- une variable de type condition (`P_valide`) est associée à chaque prédicat attendu (`P`)
- `wait(P)` est implanté par
  `si ¬P alors P_valide.wait() fsi`
- le programmeur a la responsabilité de signaler (`P_valide.signal()`) les instants/états où P est valide

Principe

- Concevoir en termes de prédicats attendus
- Simuler cette attente de prédicats avec des variables conditions

Plan

Méthodologie (1/3) : motivation

Moniteur = réalisation d'un objet partagé

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

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

Schéma générique d'une action sur l'objet partagé

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. `signaler()` les activités qui peuvent progresser à partir de E

Méthodologie (2/3)

Etapes

1. Déterminer l'interface du moniteur
2. Énoncer informellement 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 schéma précédent
7. Vérifier que
   - l'invariant est vrai à chaque transfert du contrôle du moniteur
   - le prédicat d'acceptation est vrai quand un réveil a lieu

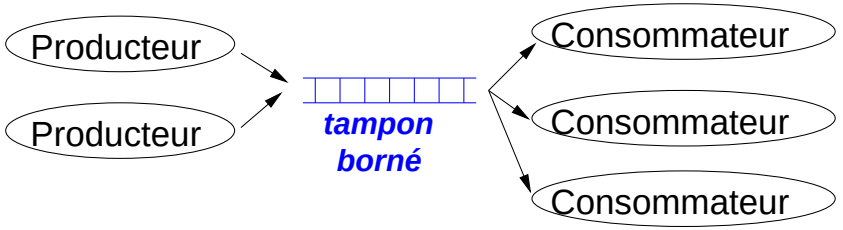
Méthodologie (3/3)

Structure standard d'une opération

si le prédicat d'acceptation est faux alors
    Attendre (*wait*) 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 (*signal*) 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).

Schéma producteurs/consommateurs



- tampon de taille borné et fixé
- nombre indéterminé et dynamique de producteurs
- " " " " de consommateurs
- objectifs : ne pas écraser une case occupée, une unique lecture consommatrice par case, attendre pour déposer si plein, attendre pour retirer si vide

Méthodologie appliquée aux producteurs/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.wait()
finsi
{ (1) nbOccupées < N }
// action applicative (ranger v dans le tampon)
nbOccupées++
{ (2) N ≥ nbOccupées > 0 }
PasVide.signal()
```

retirer(out v)

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

Vérification & Priorité

- Vérification : (2) $\Rightarrow$ (3) ? (4) $\Rightarrow$ (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
```

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` $\neq 0$ alors -- *il y a déjà un demandeur → j'attends mon tour*
    Accès.wait
finsi
si $\neg$(`nbDispo` $\geq p$) alors
    `demande` $\leftarrow p$
    `AssezDeRessources.wait` -- *au plus un bloqué ici*
    `demande` $\leftarrow 0$
finsi
`nbDispo` $\leftarrow$ `nbDispo` $- p$
Accès.signal -- au suivant de demander

libérer(q)

`nbDispo` $\leftarrow$ `nbDispo` $+ p$
si `nbDispo` $\geq$ `demande` alors
    AssezDeRessources.signal
finsi

Note : dans le cas de moniteur avec priorité au signaleur $\Rightarrow$ transformer le premier "si" de `demander` en "tant que" (suffit ici).

Plan

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 serait d'utiliser une *unique* variable condition `Accès` et d'écrire *toutes* les opérations 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

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

```
entrer(genre g)
  si nb(inv(g)) ≠ 0 alors
     accès(g).wait
  finsi
  nb(g)++
```

```
sortir(genre g)
  nb(g)--
  si nb(g) = 0 alors
     accès(inv(g)).signalAll
  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.
Contre-exemple : éviter la famine : variable `attente(genre)` pour compter les enfants en attente et ne pas accaparer la cour.

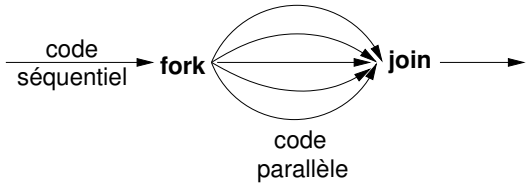
```
entrer(genre g)
  si nb(inv(g)) ≠ 0 ∨ attente(inv(g)) ≥ 4 alors
     attente(g)++
     accès(g).wait
     attente(g)--
  finsi
  nb(g)++
```

Interblocage possible avec priorité au signaleur et « tant que » à la place du « si » → repenser la solution.

Barrière

1. Barrière élémentaire : ensemble d'activités qui attendent mutuellement qu'elles soient toutes au même point (rendez-vous multiple)
2. Barrière généralisée :
   - barrière de taille M alors qu'il existe N candidats ($N > M$)
   - barrière réutilisable (cyclique) : nécessité de la refermer

Schéma de parallélisme « fork-join »



Barrière à N activités - méthodologie

1. Interface :
   - `franchir()`
2. Prédicats d'acceptation :
   - franchir() : N activités ont demandé à franchir
3. Variables d'état :
   - `nbArrivés` : natural
   - franchir() : `nbArrivés` $= N$
4. Invariant : $0 \leq$ `nbArrivés` $\leq N$
5. Variable condition : `BarrièreLevée`

Barrière à N activités – opération

franchir()

```
nbArrivés++
si ¬(nbArrivés = N) alors
     BarrièreLevée.wait
finsi
{ nbArrivés = N }
BarrièreLevée.signal    // réveil en chaîne du suivant
nbArrivés--             // ou nbArrivés ← 0
```

Note : On pourrait remplacer le réveil en chaîne par :
`si nbArrivés=N alors BarrièreLevée.signalAll`

(la sémantique de SignalAll en priorité au signalé est fragile : un seul obtient l'accès exclusif, les autres attendent leur tour)

Barrière à N activités – Priorité au signaleur ?

- Correct avec priorité au signalé
- Incorrect avec priorité au signaleur :
  - $\geq N$ peuvent passer :
    Le n-ième arrive, signale, décrémente et libère l'accès exclusif ; pendant ce temps un n+1-ième est arrivé ; s'il obtient l'accès exclusif avant celui signalé ⇒ il passe et signale ; etc. Puis tous ceux signalés passent.
  - Remplacement du `si` en `tant que` : un seul passe :
    Le n-ième arrive, signale, décrémente et libère l'accès exclusif ; celui réveillé reteste la condition, trouve `nbArrivés` à $N-1$ se rebloque.

La condition de réveil (il y a eu N arrivées) est plus faible que la condition de passage (il y a actuellement N arrivées en attente). Retester la condition de passage est trop fort.

→ se souvenir que N activités sont en cours de franchissement.

Barrière à N activités – opération

franchir(), priorité au signaleur

```
tant que (nbArrivés = N) alors
    // barrière en cours de vidage
    BarrièreBaissée.wait
fintq

nbArrivés++
tant que ¬(nbArrivés = N) alors
     BarrièreLevée.wait
fintq
si nbArrivés = N ∧ nbSortis = 0 alors // dernier arrivé
    BarrièreLevée.signalAll
finsi

nbSortis++
si nbSortis = N alors      // dernier sorti
    nbSortis ← 0
    nbArrivés ← 0
    BarrièreBaissée.signalAll
finsi
```


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 $\rightarrow$
  - raisonnement simplifié
  - meilleure lisibilité

Limites

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

Cinquième partie

Programmation multiactivité

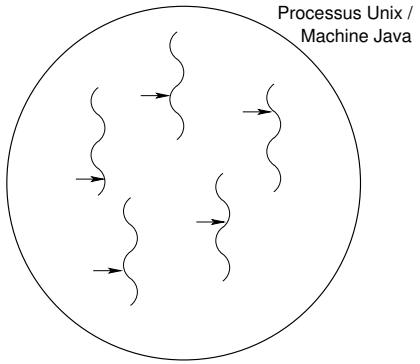
Contenu de cette partie

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

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

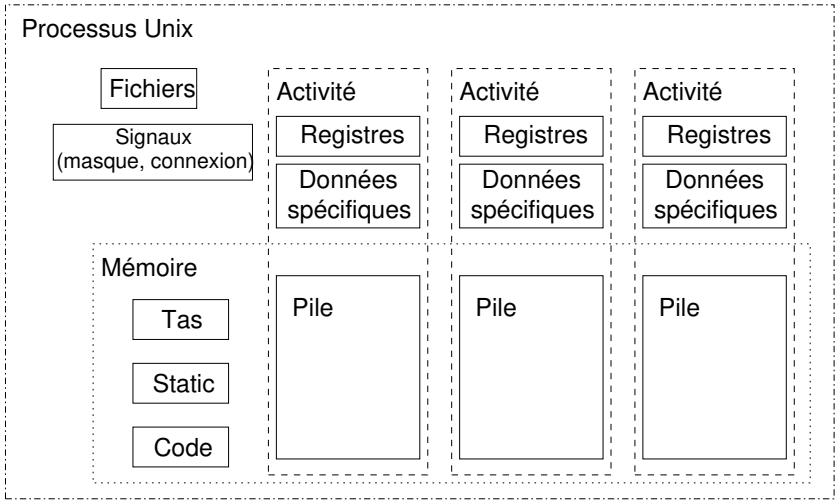
Plan

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.

Processus multiactivité



Processus lourds vs activités

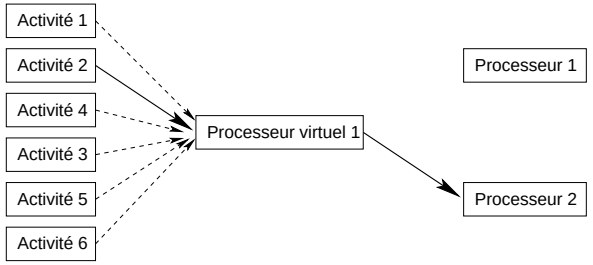
- *Processus lourds* : représentent l'exécution d'une application, du point de vue du système
  - → unité d'allocation de ressources
  - isolation des espaces d'adressage
  - commutation coûteuse
- *Activités* (threads, processus légers...) :
  - unité d'exécution : décomposition d'un traitement en sous-traitements 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
  - une bibliothèque applicative gère le partage entre activités du temps processeur alloué au processus lourd
    → commutation plus efficace.

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.

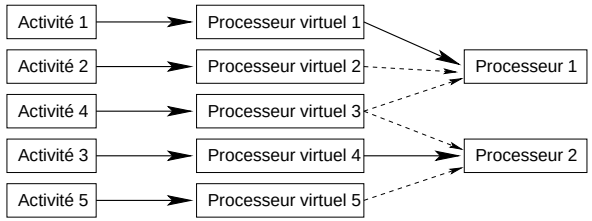
1. Many-to-one : 1 seul processeur virtuel par processus
2. Many-to-many : 1 processeur virtuel par activité
3. Many-to-few : quelques processeurs virtuels par processus

Many-to-one



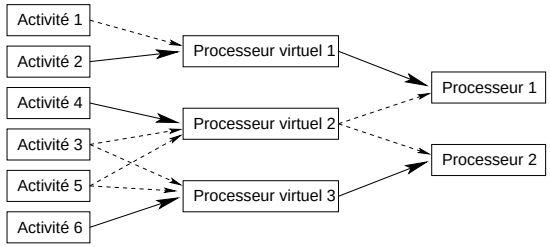
+ commutation entre activités efficace
+ implantation simple et portable
– pas de bénéfice si plusieurs processeurs
– blocage du processus (donc de toutes les activités) en cas d'appel système bloquant, ou implantation complexe

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

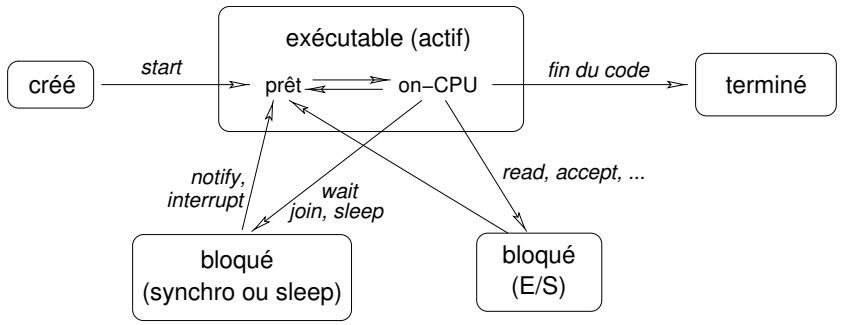
Plan

Conception d'applications parallèles en Java

Java permet de manipuler
- les processus lourds : classes `java.lang.ProcessBuilder` et `java.lang.Process`
- les activités : 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 globale propre 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
  - compteur
  - sémaphore
  - ...
- 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();
    }
    ...
```

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;
    }
}
```

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)

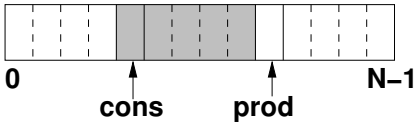
Sémaphores

Définition

Un sémaphore encapsule un entier, avec une contrainte de positivité, et deux opérations atomiques d'incrémentation et de décrémentation.

- contrainte de positivité : l'entier est toujours positif ou nul.
- opération `down` ou `acquire` : décrémente le compteur s'il est strictement positif ; si le compteur est nul, l'activité appelante est bloquée en attendant de pouvoir le décrémenter.
- opération `up` ou `release` : incrémente le compteur.

Producteurs/consommateurs avec sémaphore




```
init () { mutex := new Semaphore(1);
  vide  := new Semaphore(N); // compte le nb de cases vides
  plein  := new Semaphore(0); // compte le nb de cases occupées
}
```

```
déposer(Object v) {
  vide.acquire();
  // précond : pas tout plein
  mutex.acquire(); // excl. mutuelle
  tampon[prod] = v;
  prod = prod + 1 % N;
  mutex.release();
  // postcond : pas tout vide
  plein.release();
}
```

```
Object retirer() {
  plein.acquire();
  // précond : pas tout vide
  mutex.acquire(); // excl. mutuelle
  Object res = tampon[cons];
  cons = cons + 1 % N;
  mutex.release();
  // postcond : pas tout plein
  vide.release();
  return res;
}
```

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 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());
   }
 }
}
```

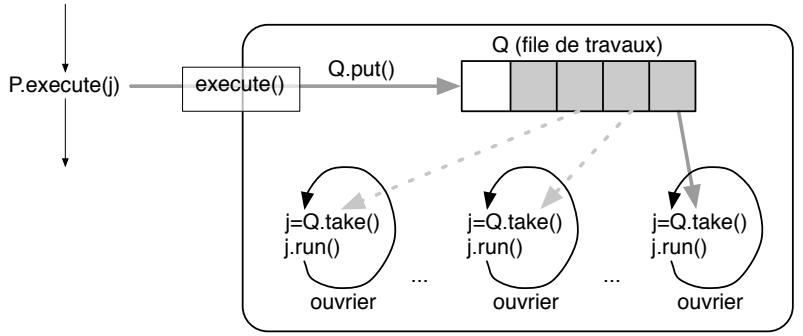
Utilisation d'un Executor (avec lambda)

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


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)]*

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();
      }
  }}
}
```


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

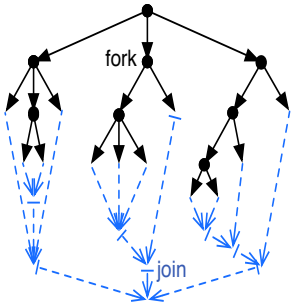
- Evaluation paresseuse : l'appel effectif d'une fonction peut être différée (éventuellement exécutée en parallèle avec l'appelant)
- `submit(...)` fournit à l'appelant une référence à la valeur future du résultat.
- L'appelant ne se bloque que quand il veut utiliser le résultat de l'appel, si l'évaluation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

```
class FonctionAsynchrone implements Callable<TypeRetour> {
  public TypeRetour call() { ... return v; }
}

ExecutorService executor = Executors.newCachedThreadPool();
Callable<TypeRetour> fonc = new FonctionAsynchrone();
Future<TypeRetour> appel = executor.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
  }
```

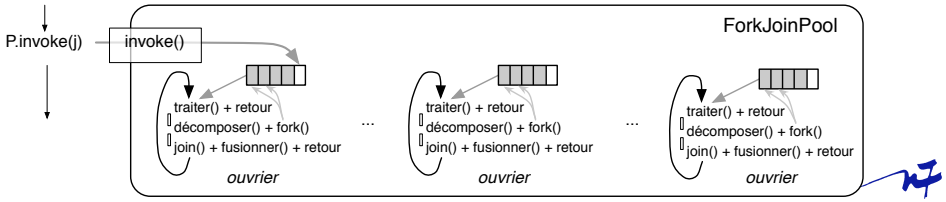


Exécuteur pour le schéma fork/join (1/3)

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)

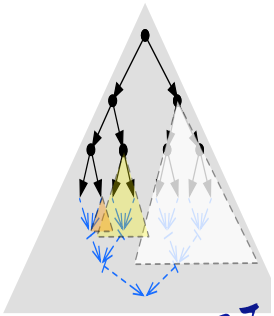
Activité d'un ouvrier du ForkJoinPool :

- 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

→

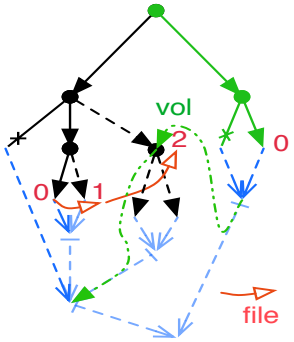
Chaque ouvrier traite un arbre de tâches qu'il

- 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 !

- programmation difficile

→ 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

Plan

PThreads : activités en C, C++

Librairie multiactivité pour le C

- manipulation d'activités (création, terminaison...) : `pthread_create`, `pthread_exit`
- synchronisation : verrous, variables condition :
  - `pthread_mutex_init`, `pthread_mutex_lock`, `pthread_mutex_unlock`
  - `pthread_cond_init`, `pthread_cond_wait`, `pthread_cond_signal`, , `pthread_cond_broadcast`
- primitives annexes : données spécifiques à chaque activité, politique d'ordonnancement...

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).

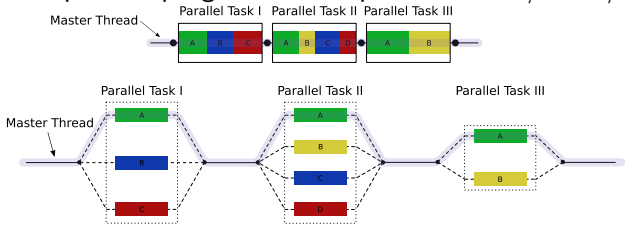
.NET (C#)

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;
```

(figure : source wikicommons)

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)

Conclusion

- Cycle de vie des activités
  - explicite
  - implicite par gestionnaire du parallélisme
- Synchronisation
  - Outil générique : moniteur (= exclusion mutuelle + conditions)
  - Objets spécifiques : producteur/consommateur, compteurs...

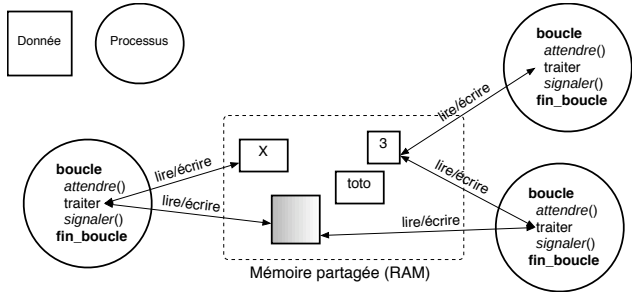
Septième partie

Processus communicants

Contenu de cette partie

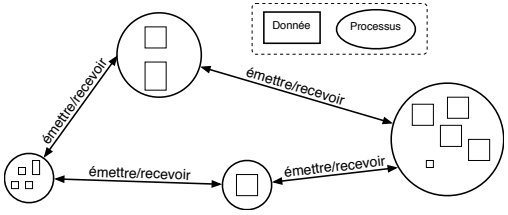
- Modèles de programmation concurrente
- Modèle des processus communicants
- Approche CSP/Go pour la programmation concurrente
  - Goroutine et canaux
  - Communiquer explicitement plutôt que partager implicitement

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

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 ≈ appel de procédure

- Émission asynchrone ⇒ buffers (messages émis non reçus)
- Synchrone ⇒ 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
```

Nommage 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)
```

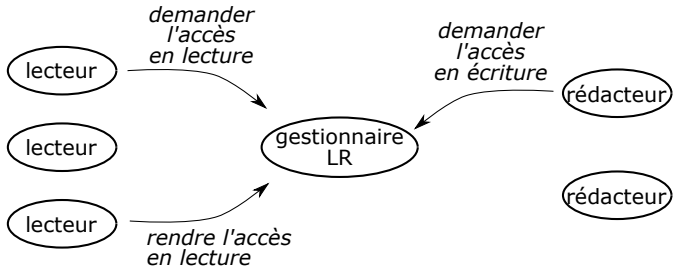
Émission bloquante → émission non bloquante

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 gestionnaire d'un objet partagé

Interactions avec l'objet partagé

Pour chaque opération faisable sur l'objet :

- émettre un message de requête vers le gestionnaire
- attendre le message de réponse de gestionnaire

Schéma de fonctionnement du gestionnaire

- 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
  $\Rightarrow$ 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...)

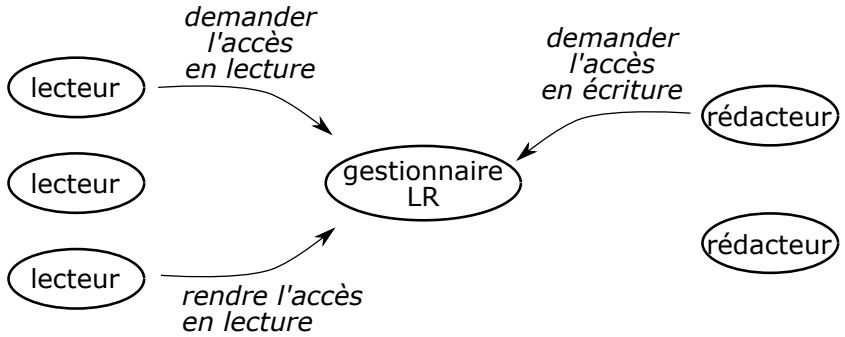
Plan

Synchronisation pure, approche condition

Pour un problème de synchronisation pure (pas d'échange de données) :

1. Identifier l'interface = les requêtes recevables $\rightarrow$ un canal par requête
2. Identifier les conditions d'acceptation pour chaque canal
3. Activité serveur qui boucle, lisant un message parmi ceux dont la condition d'acceptation est vraie (et bloquant s'il n'y a aucun tel message).

Exemple : lecteurs/rédacteurs (1/3)



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

Exemple : lecteurs/rédacteurs (2/3)

Utilisateur

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


Exemple : lecteurs/rédacteurs (3/3)

Goroutine de synchronisation

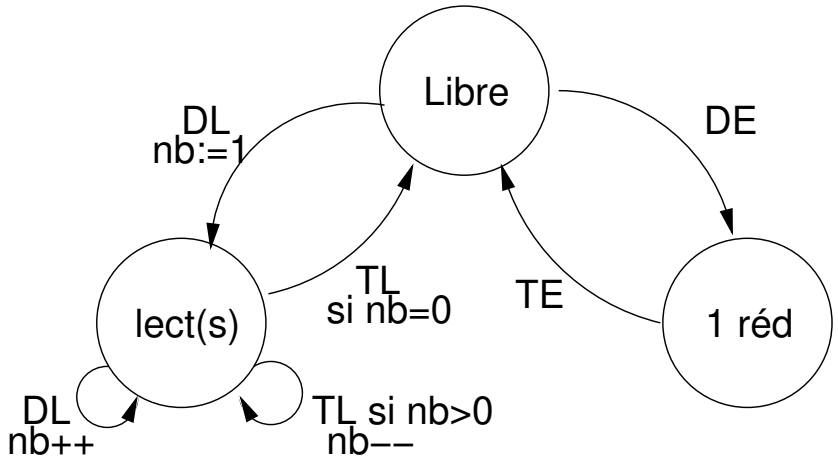
```
func when(b bool, c chan struct{}) chan struct{} {
    if  b { return c } else { return nil }
}

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--;
     }
}}
```


Synchronisation pure, approche par automate

Pour un problème de synchronisation pure (pas d'échange de données) :

1. Identifier l'interface = les requêtes recevables → un canal par requête
2. Construire un automate fini à états.
   Chaque état se distingue par les canaux acceptables en lecture.
3. L'activité serveur boucle, et selon l'état courant détermine quels messages peuvent être pris.

Exemple : lecteurs/rédacteurs (1/2)



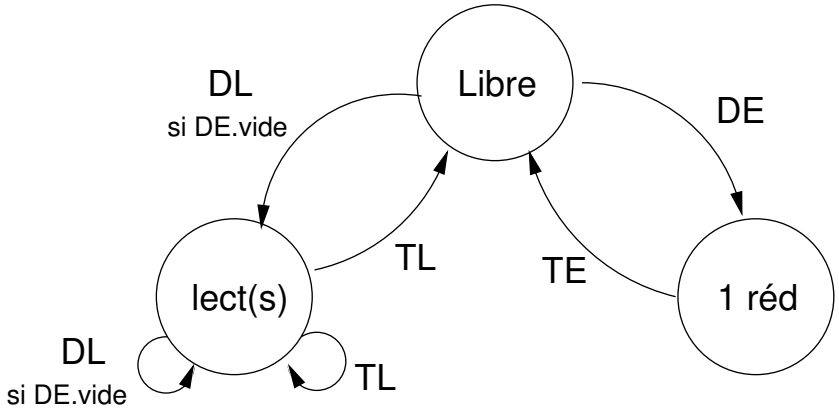
Automate étendu avec une variable d'état `nb` qui contrôle les transitions.

Exemple : Lecteurs/rédacteurs (2/2)

Goroutine de synchronisation

```
const ( lecteur = iota, redacteur, libre )
func SynchroLR() {
  nblec := 0;
  etat := libre ;
  for {
    if etat == libre {
      select {
        case <− DE : etat = redacteur
        case <− DL : etat = lecteur; nblec++ }
    } else if etat == lecteur {
      select {
        case <− DL : etat = lecteur; nblec++
        case <− TL : nblec−−; if nblec == 0 { etat = libre } }
    } else { // etat == redacteur
      select {
        case <− DE : etat = libre }
    }
  }
}
```

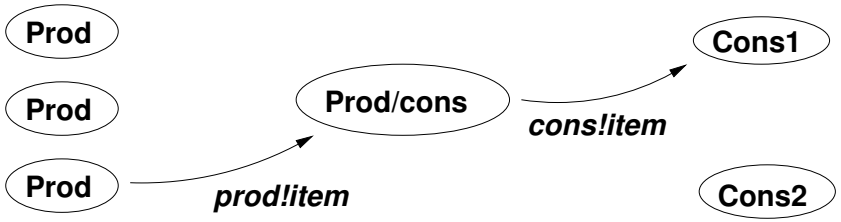

Lecteurs/rédacteurs, priorité rédacteur



`canal.Vide` est vrai s'il n'y a pas de message en attente. Toutes les variantes de CSP ne disposent pas de cette opération.

Transfert de données – architecture simpliste

Identifier l'interface en distinguant le sens de circulation de l'information : sens client → serveur ou serveur → client.



- un canal pour les demandes de dépôt (sens producteur → tampon)
- un canal pour les transmissions des valeurs déposées (sens tampon → consommateur)

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

Exemple : producteurs/consommateurs (1/2)

Producteur

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

Consommateur

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

Exemple : producteurs/consommateurs (2/2)

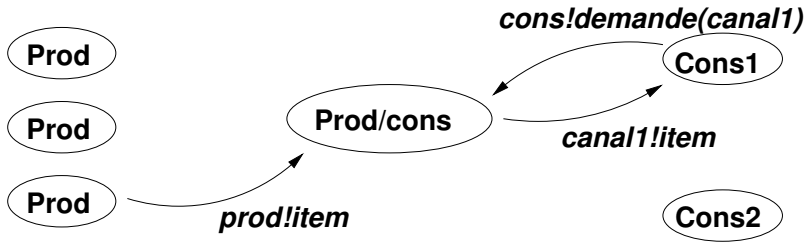
Activité de synchronisation

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

Transfert de données – Architecture évoluée

1. Un canal par type de requête
2. La requête contient, outre les paramètres de celle-ci, un canal privé au client.
3. Le serveur répond sur le canal privé quand la requête peut être satisfaite.

Exemple : Producteurs/consommateurs (1/3)



- Un canal pour les demandes de dépôt, pas de réponse attendue
- 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 (2/3)

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 (3/3)

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
  }}
```

Variables « canal », passables en paramètre et en message.

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:]
      }
    }
  }
}
```


Bilan processus communicants

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

Les langages modernes proposent la communication et synchronisation à la fois par mémoire partagée et par canaux.

Huitième partie

Synchronisation non bloquante

Plan

Limitation des verrous

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

- Interblocage : ensemble d'activités se bloquant mutuellement
- Inversion de priorité : une activité de faible priorité bloque une activité 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'une activité
- Tuer un activité ?
- 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)

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

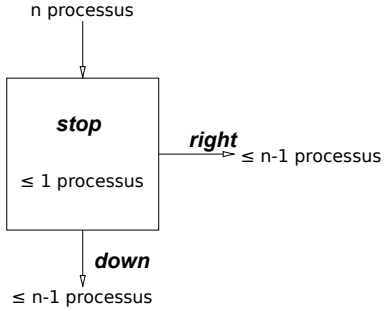
Principes

- 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



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

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

au plus $n - 1$ 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

Problème

- 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

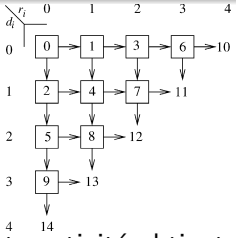
- Distributeur de numéro accédé en exclusion mutuelle
- $M = n$
- Complexité temporelle : O(1) pour un numéro, O(n) pour tous
- Une activité lente ralentit les autres

```
lock mutex; int numéro = 0;  // partagé
int obtenir_nom() {
    mutex.lock();
    int res = numéro; numéro = numéro + 1;
    mutex.unlock();
    return res;
}
```


Grille de splitters

Solution non bloquante

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



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 $n = 1$,
- 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

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

Boucle uniquement si conflit, ce n'est pas de l'attente active.

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;
```


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`

Neuvième partie

Conclusion générale

Conclusion

Programmation concurrente

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

Deux aspects

- le parallélisme
- la synchronisation

Approches

Approches traditionnelles

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

Approches modernes

- parallélisation paresseuse
- création implicite d'activités
- synchronisation implicite
- schémas classiques (fork-join)
- ne résolvent pas tous les problèmes