

Systèmes concurrents

2SN

12 octobre 2022

Matière : systèmes concurrents – organisation

Composition

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

Fonctionnement (si présentiel)

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

Evaluation

- écrit + bonus (Quiz, QCM)

Pages de l'enseignement : <http://moodle-n7.inp-toulouse.fr>,
<http://queinnec.perso.enseeiht.fr/Ens/sc.html>

Contact : queinnec@enseeiht.fr, mauran@enseeiht.fr

UE Systèmes concurrents et communicants

3 matières

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

Evaluation de l'UE

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

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

- ➊ Introduction : problématique
- ➋ Exclusion mutuelle
- ➌ Synchronisation à base de sémaphores
- ➍ Interblocage
- ➎ Synchronisation à base de moniteur
- ➏ API Java, Posix Threads
- ➐ Processus communicants – Go, Ada
- ➑ Transactions – mémoire transactionnelle
- ➒ Synchronisation non bloquante

Première partie

Introduction



6 / 38

Contenu de cette partie

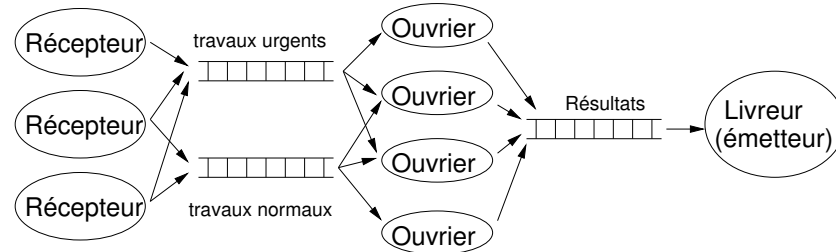
- Nature et particularités des programmes concurrents
⇒ conception et raisonnement systématiques et rigoureux
- Modélisation des systèmes concurrents
- Points clés pour faciliter la conception des applications concurrentes
- Intérêt et limites de la programmation parallèle
- Mise en œuvre de la programmation concurrente sur les architectures existantes

Plan

- ➊ Activités concurrentes
 - Le problème
 - Un peu d'architecture
 - Communication & activités
- ➋ Conception
 - Comment contrôler (réaliser) la composition ?
 - Comment décrire ?
 - Comment raisonner ?
- ➌ Avantages/inconvénients

Le 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

9 / 38

Nécessité des systèmes concurrents

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

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

11 / 38

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

10 / 38

Concurrence vs parallélisme

Parallélisme
Exécution simultanée de plusieurs codes.

Concurrence
Structuration d'un programme en activités \pm indépendantes qui interagissent et se coordonnent.

	Pas de concurrence	Concurrence
Pas de parallélisme	prog. séquentiel	plusieurs activités sur un monoprocesseur
Parallélisme	parallélisation automatique / implicite	plusieurs activités sur un multiprocesseur

12 / 38

Différence avec la programmation séquentielle

- Activités \pm simultanées \Rightarrow **explosion de l'espace d'états**

```

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

- P1 seul \rightarrow 10 états 😊
- P1 || P2 \rightarrow 10 x 10 = 100 états 😞
- P1 ; P2 \rightarrow 1 exécution 😊
- P1 || P2 \rightarrow 184756 exécutions 😞
- Interdépendance des activités
 - logique : production/utilisation de résultats intermédiaires
 - chronologique : disponibilité des résultats

\Rightarrow **non déterminisme**

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

13 / 38

Processeur

Vision simpliste : à chaque cycle, le processeur exécute une instruction machine à partir d'un flot séquentiel (le code).

En pratique :

- pipeline : plusieurs instructions en cours dans un même cycle : obtention, décodage, exécution, écriture du résultat
- superscalaire : plusieurs unités d'exécution
- instructions vectorielles
- réordonnancement (out-of-order)
- exécution spéculative

15 / 38

Composants matériels

Architecture d'un ordinateur :

- Processeurs
- Mécanisme d'interconnexion
- Mémoire et caches

14 / 38

Interconnexion

- Bus unique
 - interconnecte des processeurs entre eux
 - interconnecte les processeurs et la mémoire
 - interconnecte les processeurs et des unités d'E/S
 - médium à diffusion : usage exclusif
- Plusieurs bus dédiés
- Mini réseaux locaux, network-on-chip (parallélisme massif)
- Réseaux locaux classiques (système réparti)

16 / 38

Mémoire

La mémoire et le processeur sont éloignés : un accès mémoire est considérablement plus lent que l'exécution d'une instruction (facteur 10 à 1000 dans un ordinateur, 10 000 en réparti).

Principe de localité

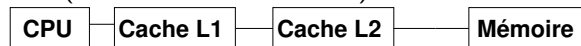
temporelle si on utilise une adresse, on l'utilisera probablement de nouveau dans peu de temps

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

⇒ conserver près du CPU les dernières cases mémoire accédées

⇒ **Cache** : mémoire rapide proche du processeur

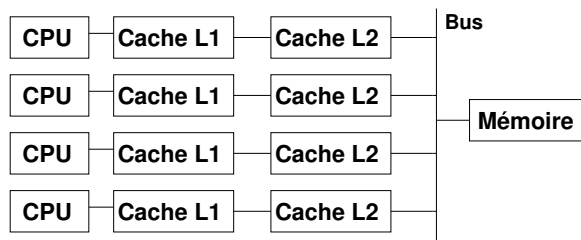
Plusieurs niveaux de caches : de plus en plus gros, de moins en moins rapides (actuellement 3 niveaux).



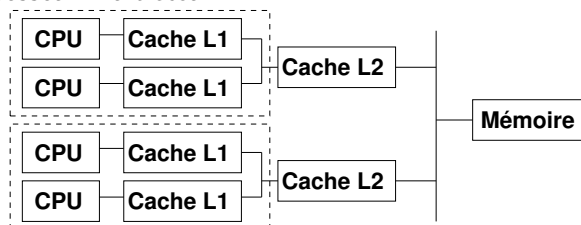
17 / 38

Architecture multiprocesseur

Multiprocesseur « à l'ancienne » :



Multiprocesseur multicœur :



19 / 38

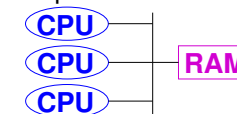
Cache

- Différents principes de placement dans le cache (associatif, mappé, k-associatif...)
- Différentes stratégies de remplacement (LRU - least recently used...)
- Différentes stratégies d'invalidation - cohérence mémoire

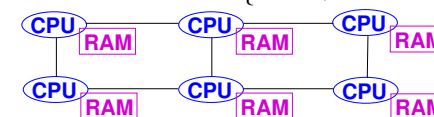
18 / 38

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

20 / 38

Écritures en mémoire

Comment fonctionne l'écriture d'une case mémoire en présence de caches ?

Write-Through diffusion sur le bus à chaque valeur écrite

- + visible par les autres processeurs \Rightarrow invalidation des valeurs passées
- + la mémoire et le cache sont cohérents
- trafic inutile : écritures répétées, écritures de variables privées au thread

Write-Back diffusion uniquement à l'éviction de la ligne

- + trafic minimal
- cohérence cache - mémoire - autres caches ?

21 / 38

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)

23 / 38

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, \dots , 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 \ll lente \gg (slow consistency) : une lecture retourne une valeur précédemment écrite, sans remonter dans le temps.

22 / 38

Activité

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

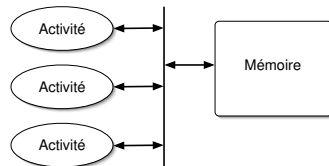
- exécution d'un programme séquentiel
- entité **logicielle**
- exécutable par un processeur
- interruptible et commutable

24 / 38

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

25 / 38

Plan

1 Activités concurrentes

- Le problème
- Un peu d'architecture
- Communication & activités

2 Conception

- Comment contrôler (réaliser) la composition ?
- Comment décrire ?
- Comment raisonner ?

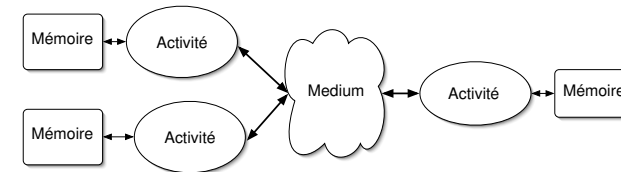
3 Avantages/inconvénients

27 / 38

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

26 / 38

Contrôler

Concevoir une application concurrente

- contrôler un ensemble d'activités concurrentes
- contrôler la **progression** et les interactions de chaque activité
- assurer leur **protection** réciproque

Moyen

attente (par blocage – suspension – de l'activité)

→ déblocage nécessairement par une autre activité

28 / 38

Contrôler

Hypothèses de bon comportement des activités : un **protocole** définit les interactions possibles :

- Opérations et paramètres autorisés.
- **Séquences d'actions autorisées.**
Un ouvrier ne doit pas déposer plus de résultats qu'il n'a pris de travaux.

29 / 38

Décrire

Triplets de Hoare

précondition/action/postcondition

Exemple

$\{t = \text{nb travaux en attente} \wedge t > 0 \wedge r = \text{nb résultats}\}$
ouvrier effectue un travail
 $\{\text{nb travaux en attente} = t - 1 \wedge \text{nb résultats} = r + 1\}$

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

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

31 / 38

Décrire

Compteurs d'événements

Compter les actions ou les changements d'états et les relier entre eux.

Exemple

Invariant $\# \text{nb de travaux soumis} = \# \text{nb travaux effectués}$
 $+ \# \text{nb travaux en cours}$
 $+ \# \text{nb travaux en attente}$

30 / 38

Décrire

Propriétés temporelles

Linéaires : pour **toutes** les exécutions possibles,
à tout moment d'une exécution.
Arborescentes : pour **certaines** exécutions possibles,
à tout moment d'une exécution.

Exemple

- Sûreté : rien de mauvais ne se produit
*Deux ouvriers ne peuvent **jamais** prendre le même travail.*
- Vivacité : quelque chose de bon finit par se produire
*Un travail déposé **fin**it **par** être pris par un ouvrier.*
- Possibilité : deux travaux déposés consécutivement **peuvent** être exécutés séquentiellement par le même ouvrier.

32 / 38

Modèle : l'entrelacement

Raisonner sur tous les cas parallèles est trop complexe
⇒ 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 $P \parallel Q$ 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

- Ne pas oublier que c'est un modèle simplificateur (vraie concurrence, cohérence mémoire...)
- Il peut ne pas exister de code séquentiel équivalent au code parallèle.

33 / 38

Plan

1 Activités concurrentes

- Le problème
- Un peu d'architecture
- Communication & activités

2 Conception

- Comment contrôler (réaliser) la composition ?
- Comment décrire ?
- Comment raisonner ?

3 Avantages/inconvénients

35 / 38

Raisonner

Contrôler les effets des interactions

- isoler (raisonner indépendamment) ⇒ modularité
- spécifier/contrôler l'interaction
- **schémas connus d'interaction** (design patterns)

34 / 38

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...

36 / 38

Parallélisme et performance

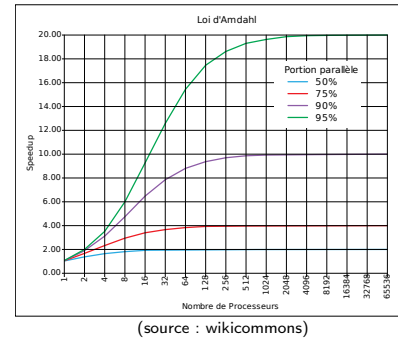
Mythe du parallélisme

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

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

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

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



(source : wikicommons)

Parallélisme et performance

Mythe de la performance

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

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

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

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

Contenu de cette partie

Deuxième partie

L'exclusion mutuelle

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



2 / 31



3 / 31

Plan

- 1 Interférences entre actions
 - Isolation
 - L'exclusion mutuelle
- 2 Mise en œuvre
 - Solutions logicielles
 - Solutions matérielles
 - Primitives du système d'exploitation
 - En pratique...



4 / 31

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



5 / 31

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



6 / 31

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
```

⇒ zéro pain possible

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
```



8 / 31

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...



7 / 31

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



9 / 31

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.



10 / 31

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 := 0x0001 > || < x := 0x0002 >
⇒ 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!

```



12 / 31

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. »



11 / 31

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



13 / 31

Propriétés du protocole d'accès



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

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

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

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

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

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

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



14 / 31

Comment ?



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

Forme générale

Variables partagées par toutes les activités

Activité A_i

entrée

section critique

sortie



16 / 31

Plan

- 1 Interférences entre actions
 - Isolation
 - L'exclusion mutuelle
- 2 Mise en œuvre
 - Solutions logicielles
 - Solutions matérielles
 - Primitives du système d'exploitation
 - En pratique...



15 / 31

Une fausse solution



Algorithme

occupé : shared boolean := false;

tant que *occupé* faire nop;

occupé ← true;

section critique

occupé ← false;

(Test-and-set non atomique)



17 / 31

Alternance



Algorithme

```

tour : shared 0..1;

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

```

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



18 / 31

Peterson 1981



Algorithme

```

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

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

```

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



20 / 31

Priorité à l'autre demandeur



Algorithme

```

demande : shared array 0..1 of boolean;

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

```

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



19 / 31

Solution pour N activités (Lamport 1974)

L'algorithme de la boulangerie

```

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

choix[i] ← true;
int tour ← 0; // local à l'activité
pour k de 0 à N faire tour ← max(tour, num[k]);
num[i] ← tour + 1;
choix[i] ← false;

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

```



21 / 31

Instruction matérielle TestAndSet



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

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

Définition

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



22 / 31

Utilisation du TestAndSet

Alors : protocole d'exclusion mutuelle :

Algorithme

```
occupé : shared boolean := false;

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

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



23 / 31

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);
```



24 / 31

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"



25 / 31

Masquage des interruptions



Éviter la **préemption** du processeur par une autre activité :

Algorithme

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

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

→ μ -système embarqué



26 / 31

Le système d'exploitation

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



27 / 31

Ordonnanceur avec priorités



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

Algorithme

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

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

→ système embarqué



28 / 31

Éviter l'attente active : contrôle des activités



Algorithme

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

bloc atomique
  si occupé alors
    self ← identifiant de l'activité courante
    ajouter self dans demandeurs
    se suspendre
  sinon
    occupé ← true
  fin si
fin bloc

section critique

bloc atomique
  si demandeurs est non vide alors
    p ← extraire premier de demandeurs
    débloquer p
  sinon
    occupé ← false
  fin si
fin bloc
```

Le système de fichiers (!)

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

Algorithme

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

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



30 / 31

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 débloquent ; sinon marquer le verrou comme libre.

Algorithme

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



31 / 31

Troisième partie

Sémaphores

Contenu de cette partie

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



2 / 30



3 / 30

Plan

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



4 / 30



But

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

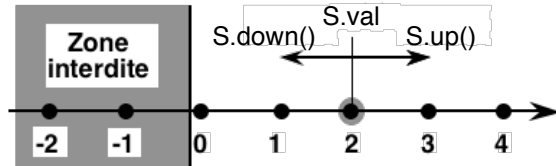


5 / 30

Définition – Dijkstra 1968

Un sémaphore S est un objet dont

- l'état val est un attribut entier *privé* (l'état est encapsulé)
- l'ensemble des états permis est contraint par un invariant (*contrainte de synchronisation*) :
invariant $S.val \geq 0$ (l'état doit toujours rester positif ou nul)
- l'interface fournit deux opérations principales :
 - *down* : décrémente l'état s'il est > 0 , **bloque** si l'état est nul,
 - *up* : incrémente l'état \rightarrow peut **débloquer une** éventuelle activité bloquée sur *down*
 - les opérations *down* et *up* sont **atomiques**



6 / 30

Modèle intuitif

Un sémaphore peut être vu comme un tas de jetons avec deux opérations :

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

Attention

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

8 / 30

- *Autre opération* : constructeur (et/ou initialisation)

$S = \text{new Semaphore}(v_0)$ (ou $S.\text{init}(v_0)$)
(crée et) initialise l'état de S à v_0

- *Autres noms des opérations*

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

7 / 30

Spécification formelle : Hoare

Définition

Un sémaphore S encapsule un entier val tel que

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

Remarques

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

9 / 30

Remarques

- 1 Lors de l'exécution d'une opération *up*, s'il existe plusieurs activités en attente, la politique de choix de l'activité à débloquer peut être :
 - par ordre chronologique d'arrivée (FIFO) : équitable
 - associée à une priorité affectée aux activités en attente
 - indéfinie.

C'est le cas le plus courant : avec une primitive rapide mais non équitable, on peut implanter (laborieusement) une solution équitable, mais avec une primitive lente et équitable, on **ne peut pas** implanter une solution rapide.
- 2 Variante : *down* non bloquant (*tryDown*)

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

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



10 / 30

Plan

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



12 / 30

Sémaphore binaire (booléen) – Verrou

Définition

Sémaphore *S* encapsulant un entier *b* tel que

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

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



11 / 30

Schémas d'utilisation essentiels (0/4)

Réalisation de l'isolation : exclusion mutuelle

Algorithme

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

// Protocole d'exclusion mutuelle
// (suivi par chacune des activités)

    section critique
```



13 / 30

Schémas d'utilisation essentiels (0/4)

Réalisation de l'isolation : exclusion mutuelle

Algorithme

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

// Protocole d'exclusion mutuelle
// (suivi par chacune des activités)

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



14 / 30

Schémas d'utilisation essentiels (2/4)

Synchronisation élémentaire : attendre/signaler un événement E

- Objet partagé :
occurrenceE = new Semaphore(0) // initialisé à 0
- attendre une occurrence de E : occurrenceE.down()
- signaler l'occurrence de l'événement E : occurrenceE.up()

Règle de conception

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



16 / 30

Schémas d'utilisation essentiels (1/4)

Généralisation : contrôle du degré de parallélisme

Algorithme

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

- Objet partagé :
global accèsR = new Semaphore(Max)
- Protocole d'accès à la ressource R (pour *chaque* activité) :
accèsR.down()
accès à la ressource R
accèsR.up()

Règle de conception

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



15 / 30

Schémas d'utilisation essentiels (3/4)

Synchronisation élémentaire : rendez-vous entre deux activités

Problème : garantir l'exécution « virtuellement » simultanée d'un point donné du flot de contrôle de A et d'un point donné du flot de contrôle de B

- Objets partagés :
aArrivé = new Semaphore(0);
bArrivé = new Semaphore(0) // initialisés à 0
- Protocole de rendez-vous :

<i>Activité A</i>	<i>Activité B</i>
...	...
aArrivé.up()	bArrivé.up()
bArrivé.down()	aArrivé.down()
...	...



17 / 30

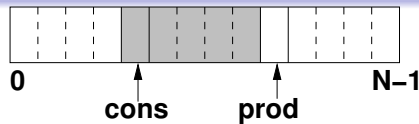
Schémas d'utilisation essentiels (4/4)

Généralisation : rendez-vous à N activités (barrière)

Fonctionnement : pour passer la barrière, une activité doit attendre que les $N - 1$ autres activités l'aient atteinte.

- Objet partagé :
barrière = tableau $[0..N-1]$ de Semaphore;
pour $i := 0$ à $N-1$ faire barrière[i].init(0) finpour;
- Protocole de passage de la barrière (pour l'activité i) :
pour $k := 0$ à $N-1$ faire
 barrière[i].up()
finpour;
pour $k := 0$ à $N-1$ faire
 barrière[k].down()
finpour;

18 / 30



producteur

```
produire(i) {i : Item}
libre.down()
{ pré : ∃ places libres }
mutex.down()
{ pré : début SC }
{ dépôt dans le tampon }
tampon[prod] := i
prod := prod + 1 mod N
{ post : fin SC }
mutex.up()
{ post : ∃ places occupées }
occupé.up()
```

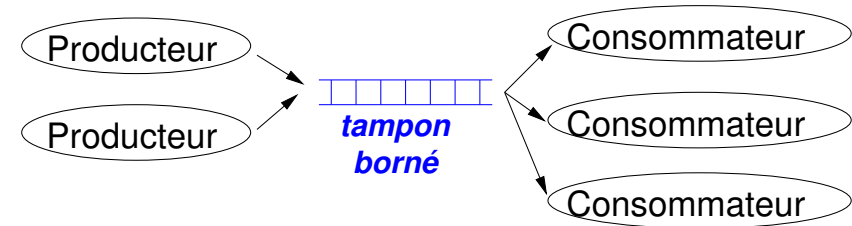
consommateur

```
occupé.down()
{ pré : ∃ places occupées }
mutex.down()
{ pré : début section critique }
{ retrait du tampon }
i := tampon[cons]
cons := cons + 1 mod N
{ post : fin SC }
mutex.up()
{ post : ∃ places libres }
libre.up()
consommer(i) {i : Item}
```

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

20 / 30

Schéma producteurs/consommateurs : tampon borné



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

19 / 30

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

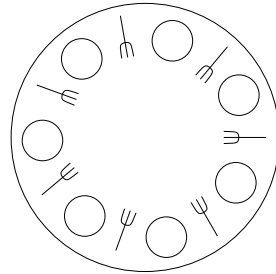
- N ressources critiques, équivalentes, réutilisables
- usage exclusif des ressources
- opération **allouer** $k \leq N$ ressources
- opération **libérer** des ressources précédemment obtenues
- bon comportement :
 - pas deux demandes d'allocation consécutives sans libération intermédiaire
 - une activité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

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

21 / 30

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

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



Un philosophe peut être :

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

Allocation multiple de ressources différenciées, interblocage...

22 / 30

Plan

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

24 / 30

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

Une ressource peut être utilisée :

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

Souvent rencontré sous la forme de **verrou lecture/écriture** (read-write lock).

Permet l'isolation des modifications avec un meilleur parallélisme que l'exclusion mutuelle.

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

23 / 30

Implantation d'un sémaphore

Repose sur un service de gestion des processus fournissant :

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

Implantation

```
Sémaphore = { int nbjetons ;
               File<Processus> bloqués }
```

25 / 30

Algorithme

```

S.down() = entrer en excl. mutuelle
           si S.nbjets = 0 alors
               insérer self dans S.bloqués
               suspendre le processus courant
           sinon
               S.nbjets ← S.nbjets - 1
           finsi
           sortir d'excl. mutuelle

S.up() =   entrer en excl. mutuelle
           si S.bloqués ≠ vide alors
               procRéveillé ← extraire de S.bloqués
               débloquent procRéveillé
           sinon
               S.nbjets ← S.nbjets + 1
           finsi
           sortir d'excl. mutuelle

```

26 / 30

Compléments (1/3) :

réalisation d'un sémaphore général à partir de sémaphores binaires

```

Sg = ⟨ val := ?,
      mutex = new SemaphoreBinaire(1),
      accès = new SemaphoreBinaire(val > 0; 1; 0) // verrous
    ⟩

Sg.down() = Sg.accès.down()
            Sg.mutex.down()
            S.val ← S.val - 1
            si S.val ≥ 1 alors Sg.accès.up()
            Sg.mutex.up()

Sg.up() =   Sg.mutex.down()
            S.val ← S.val + 1
            si S.val = 1 alors Sg.accès.up()
            Sg.mutex.up()

```

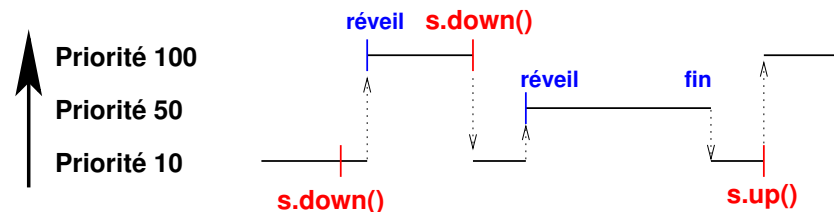
→ les sémaphores binaires ont (au moins) la même puissance d'expression que les sémaphores généraux

27 / 30

Compléments (2/3) : sémaphores et priorités

Temps-réel ⇒ priorité ⇒ sémaphore non-FIFO.

Inversion de priorités : une activité moins prioritaire bloque/retarde indirectement une activité plus prioritaire.



28 / 30

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

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

29 / 30

Conclusion

Les sémaphores

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



Quatrième partie

Interblocage

Contenu de cette partie

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



2 / 25



Plan

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



4 / 25



3 / 25

Allocation de ressources multiples

But : gérer la compétition entre activités

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

Modèle/protocole « général »

- Ressources banalisées, réutilisables, identifiées
- Ressources regroupées en classes :
une activité peut demander des ressources dans chaque classe
- Ressources allouées par un **gérant de ressources**
 - **demander** (IdClasse → naturel) → (Set of IdRessource)
demander est **bloquant** si des ressources manquent.
 - **libérer** (Set of IdRessource) → unit
- Le gérant :
 - rend les ressources libérées utilisables par d'autres activités
 - libère les ressources détenues, à la terminaison d'une activité.



5 / 25



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

- **Vivacité faible (progression)** : si **des** activités déposent des requêtes continûment, **une** des requêtes finira par être satisfaite ;
- **Vivacité forte (équité faible)** : si une activité dépose sa requête de manière continue, elle finira par être satisfaite ;
- **Équité forte** : si une activité dépose une requête **infiniment souvent**, elle finira par être satisfaite ;
- **Équité FIFO** : si une activité dépose une requête, elle sera satisfaite avant toute autre requête (conflictuelle) déposée par la suite.

Famine (privation)

Une activité est en **famine** lorsqu'elle attend infiniment longtemps la satisfaction de sa requête (elle n'est jamais satisfaite).

6 / 25

Le problème

Contexte : allocation de ressources réutilisables

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

Problème

P_1 demande A puis B ,

P_2 demande B puis A

→ risque d'interblocage :

- 1 P_1 demande et obtient A
- 2 P_2 demande et obtient B
- 3 P_2 demande A → se bloque
- 4 P_1 demande B → se bloque

8 / 25

Plan

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

7 / 25

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.

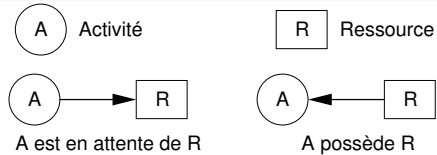
Pour l'ensemble d'activités interbloquées :
Interblocage \equiv négation de la progression

absence de famine \Rightarrow absence d'interblocage

L'interblocage est un état stable.

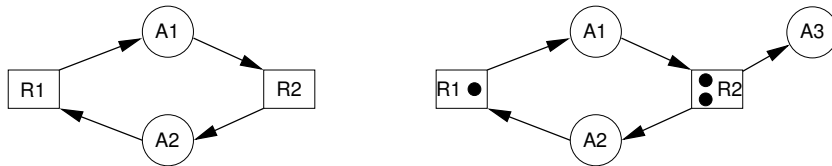
9 / 25

Notation : graphe d'allocation



Condition nécessaire et suffisante à l'interblocage

Attente circulaire (cycle/knot dans le graphe d'allocation)



Solutions

Prévention : empêcher la formation de cycles dans le graphe

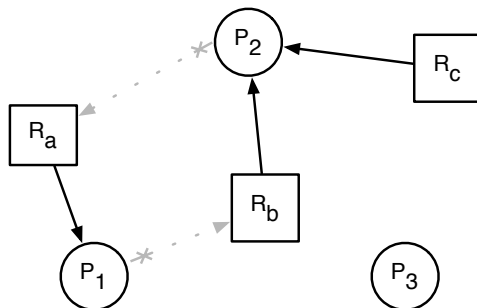
Détection + guérison : détecter l'interblocage, et l'éliminer

10 / 25

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

Éviter le blocage des activités

→ pas d'attente → pas d'arcs sortant d'une activité



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

12 / 25

Plan

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

11 / 25

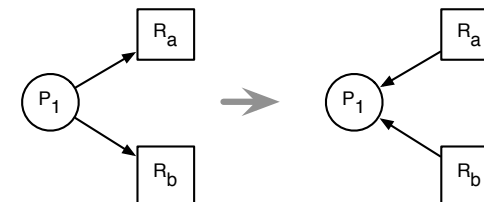
Comment éviter par construction la formation de cycles ? (2/4)

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



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

13 / 25

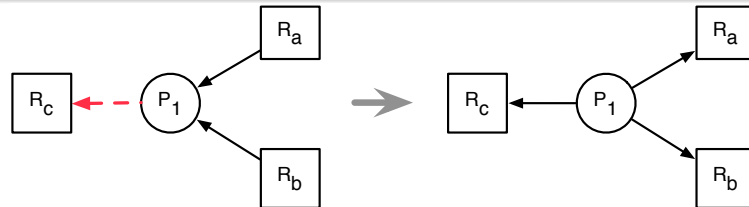
Comment éviter par construction la formation de cycles ? (3/4)

Permettre la réquisition des ressources allouées

- éliminer/inverser les arcs entrants d'une activité en cas de création d'arcs sortants

Une activité bloquée doit

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

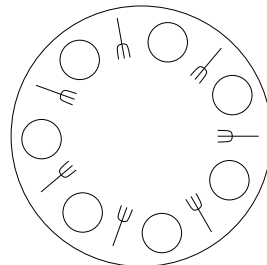


(optimisation : restitution paresseuse des ressources : libération que si la demande est bloquante)

14 / 25

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 :

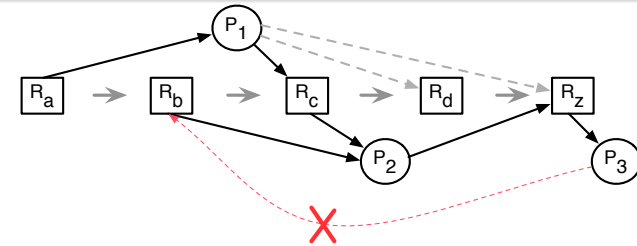
- penseur : il n'utilise pas de fourchettes ;
- mangeur : il utilise les deux fourchettes adjacentes ; aucun de ses voisins ne peut manger ;
- demandeur : il souhaite manger mais ne dispose pas des deux fourchettes.

16 / 25

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

Fixer un ordre global sur les demandes : classes ordonnées

- un **ordre** est défini **sur les ressources**
- toute activité doit demander les ressources en suivant 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
(car un cycle est un chemin sur lequel l'ordre des ressources n'est pas respecté)

15 / 25

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.

17 / 25

Esquive

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

L'algorithme du banquier

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

18 / 25

Algorithme du banquier (1/2)

Allocation de Demande ressources au processus IdProc

```

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

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

```

20 / 25

Algorithme du banquier : exemple

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

	max	poss.	dem	
P_0	10	5		
P_1	4	2	+1	oui
				✓ $(5 + 4 + 2 \leq 12)$
				✓ $(10 + (0) + 2 \leq 12)$
				✓ $((0) + (0) + 9 \leq 12)$
P_2	9	2	+1	non
				✗ $(10 + 2 + 3 > 12)$
				✗ $(5 + 2 + 9 > 12)$
				✓ $(5 + 4 + 3 \leq 12)$
				✗ $(10 + (0) + 3 > 12)$
				✗ $(5 + (0) + 9 > 12)$

19 / 25

Algorithme du banquier (2/2)

```

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

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

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

```

21 / 25

Plan

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



22 / 25



Plan

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



24 / 25



Détection

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

Coûteux → exécution périodique (et non à chaque allocation)

Guérison : Réquisition des ressources allouées à une/des activités interbloquées

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



23 / 25

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



25 / 25

Cinquième partie

Moniteurs

Contenu de cette partie

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



2 / 40



Plan

- 1 Introduction
- 2 Définition
 - Notion de moniteur Hoare, Brinch Hansen 1973
 - Expression de la synchronisation : type « condition »
 - Exemple
 - Transfert du contrôle exclusif
- 3 Utilisation des moniteurs
 - Méthodologie
 - Exemple : producteurs/consommateurs
- 4 Conclusion
- 5 Annexes
 - Allocateur de ressources
 - Variantes
 - Réveil multiple
 - Priorité au signalé/signaleur
 - Barrière



4 / 40



Limites des sémaphores

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

Exemples

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



3 / 40



5 / 40

Plan

- 1 Introduction
- 2 Définition
 - Notion de moniteur Hoare, Brinch Hansen 1973
 - Expression de la synchronisation : type « condition »
 - Exemple
 - Transfert du contrôle exclusif
- 3 Utilisation des moniteurs
 - Méthodologie
 - Exemple : producteurs/consommateurs
- 4 Conclusion
- 5 Annexes
 - Allocateur de ressources
 - Variantes
 - Réveil multiple
 - Priorité au signalé/signaleur
 - Barrière

6 / 40

Expression de la synchronisation : type *condition*

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

- Une *file d'attente* est associée à *chaque* variable condition
- Opérations possibles sur une variable de type condition *C* :
 - *C.attendre()* [*C.wait()*] : bloque et range dans la file associée à *C* l'activité appelante, puis libère l'accès exclusif au moniteur.
 - *C.signaler()* [*C.signal()*] : si des activités sont bloquées sur *C*, en réveille une ; sinon, opération nulle.
- *condition* \approx *événement*
 - *condition* \neq *sémaphore* (pas de mémoire des appels à *C.signaler()*)
 - *condition* \neq *prédicat logique*
- autres opérations sur les conditions :
 - *C.vide()* : renvoie vrai si aucune activité n'est bloquée sur *C*
 - *C.attendre(priorité)* : réveil des activités bloquées sur *C* selon une priorité

8 / 40

Notion de moniteur 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 *procédures* (*opérations*)

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

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

7 / 40

Exemple : travail délégué

Schéma client/serveur asynchrone : 1 client + 1 serveur

Les activités

Client	Serveur
<i>boucle</i>	<i>boucle</i>
:	:
déposer_travail(t)	x ← prendre_travail()
:	// (y ← f(x))
r ← lire_résultat()	rendre_résultat(y)
:	:
<i>fin_boucle</i>	<i>fin_boucle</i>

9 / 40

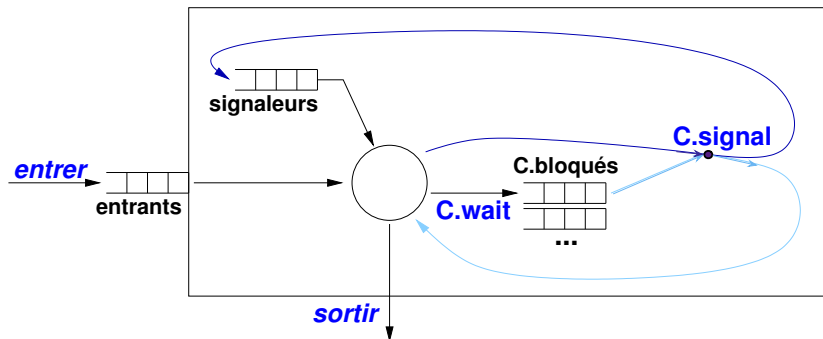
Exemple – le moniteur

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

<pre>déposer_travail(in t) { (pas d'attente) req ← t Dépôt.signaler() }</pre>	<pre>prendre_travail(out t) si req = null alors Dépôt.attendre() finssi t ← req req ← null {RAS}</pre>
<pre>lire_résultat(out r) si rés = null alors Dispo.attendre() finssi r ← rés rés ← null {RAS}</pre>	<pre>rendre_résultat(in y) { (pas d'attente) rés ← y Dispo.signaler() }</pre>

10 / 40

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)

12 / 40

Transfert du contrôle exclusif

Les opérations du moniteur s'exécutent en exclusion mutuelle.

→ Lors d'un **réveil** par *signaler()*, **qui** obtient l'accès exclusif?

Priorité au signalé

Lors du réveil par *signaler()*,

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

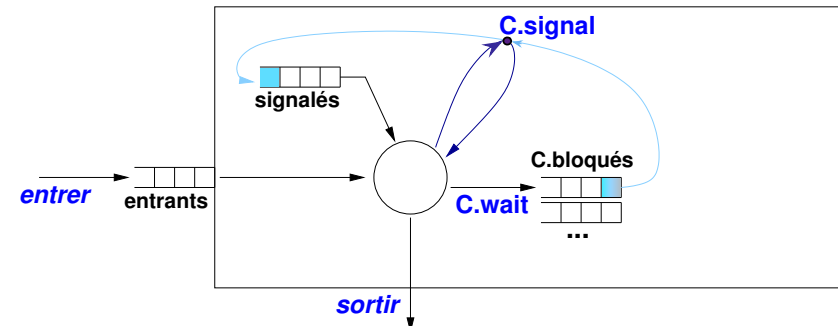
Priorité au signaleur

Lors du réveil par *signaler()*,

- 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

11 / 40

Priorité au signaleur avec file spécifique des signalés

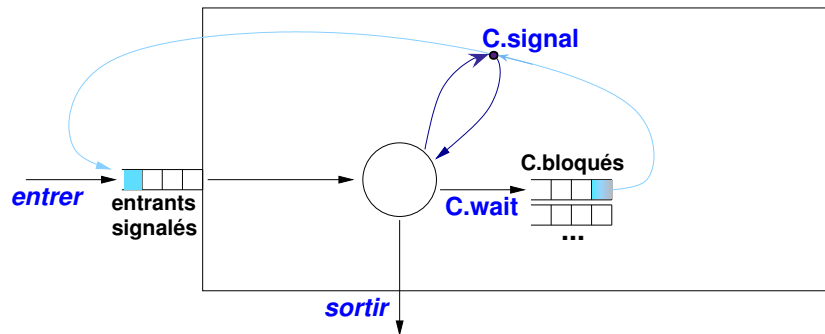


C.signal()

- 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

13 / 40

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

14 / 40

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 réveill obtient le moniteur **ultérieurement**, éventuellement après d'autres activités
 - Implantation du mécanisme plus simple et plus performante
 - Au réveil, le signalé doit **retester la condition de déblocage**
 - Possibilité de famine, écriture et raisonnements plus lourds

16 / 40

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

Priorité au signalé

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

Priorité au signaleur

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

15 / 40

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

Idée (d'origine)

Attente sur des **prédicats**,

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

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

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

variables d'état : req, rés --Requête/Résultat en attente (null si aucun(e))	
entrée <code>déposer_travail(in t)</code> <code>req ← t</code>	entrée <code>prendre_travail(out t)</code> <code>attendre(req ≠ null)</code> <code>t ← req</code> <code>req ← null</code>
entrée <code>lire_résultat(out r)</code> <code>attendre(rés ≠ null)</code> <code>r ← rés</code> <code>rés ← null</code>	entrée <code>rendre_résultat(in y)</code> <code>rés ← y</code>

17 / 40

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

Efficacité problématique :

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

→ gestion de l'évaluation laissée au programmeur

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

Principe

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

18 / 40

Méthodologie (1/3) : motivation

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

- permet de concevoir la synchronisation en termes d'interactions entre chaque activité et **un** objet partagé :
les seules interactions autorisées sont celles qui laissent l'objet partagé dans un état cohérent
- **Invariant du moniteur** = ensemble des états possibles pour l'objet géré par le moniteur

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

- 1 si l'exécution de A (depuis l'état courant) invalide I
alors $attendre()$ **fin**si { **prédicat d'acceptation** de A }
- 2 effectuer A { → **nouvel état courant E** }
- 3 **réveiller()** les activités qui peuvent progresser à partir de E

20 / 40

Plan

- 1 Introduction
- 2 Définition
 - Notion de moniteur Hoare, Brinch Hansen 1973
 - Expression de la synchronisation : type « condition »
 - Exemple
 - Transfert du contrôle exclusif
- 3 Utilisation des moniteurs
 - Méthodologie
 - Exemple : producteurs/consommateurs
- 4 Conclusion
- 5 Annexes
 - Allocateur de ressources
 - Variantes
 - Réveil multiple
 - Priorité au signalé/signaleur
 - Barrière

19 / 40

Méthodologie (2/3)

Etapes

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

21 / 40

Méthodologie (3/3)

Structure standard d'une opération

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

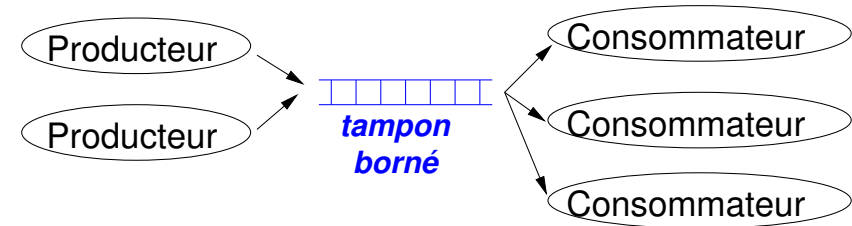
Vérifier, pour chaque variable condition, que
 chaque précondition de *signaler()* (2)
 implique chaque postcondition de *attendre()* (1)

22 / 40

- 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 : $\text{nbOccupées} < N$
 - retirer : $\text{nbOccupées} > 0$
- 4 Invariant : $0 \leq \text{nbOccupées} \leq N$
- 5 Variables conditions : PasPlein, PasVide

24 / 40

Exemple : réalisation du schéma producteurs/consommateurs



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

23 / 40

déposer(in v)

```

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

retirer(out v)

```

si  $\neg(\text{nbOccupées} > 0)$  alors
  PasVide.attendre()
fin si
{ (3)  $\text{nbOccupées} > 0$  }
// action applicative (prendre v dans le tampon)
nbOccupées --
{ (4)  $0 \leq \text{nbOccupées} < N$  }
PasPlein.signaler()
  
```

25 / 40

Vérification & Priorité

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

```

déposer(in v)
  tant que  $\neg(\text{nbOccupées} < N)$  faire
    PasPlein.wait
  fintq
  { (1)  $\text{nbOccupées} < N$  }
  // action applicative (ranger v dans le tampon)
  nbOccupées ++
  { (2)  $N \geq \text{nbOccupées} > 0$  }
  PasVide.signal
    
```

26 / 40

Conclusion

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

Apports

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

Limites

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

28 / 40

Plan

- 1 Introduction
- 2 Définition
 - Notion de moniteur Hoare, Brinch Hansen 1973
 - Expression de la synchronisation : type « condition »
 - Exemple
 - Transfert du contrôle exclusif
- 3 Utilisation des moniteurs
 - Méthodologie
 - Exemple : producteurs/consommateurs
- 4 Conclusion
- 5 Annexes
 - Allocateur de ressources
 - Variantes
 - Réveil multiple
 - Priorité au signalé/signaleur
 - Barrière

27 / 40

Plan

- 1 Introduction
- 2 Définition
 - Notion de moniteur Hoare, Brinch Hansen 1973
 - Expression de la synchronisation : type « condition »
 - Exemple
 - Transfert du contrôle exclusif
- 3 Utilisation des moniteurs
 - Méthodologie
 - Exemple : producteurs/consommateurs
- 4 Conclusion
- 5 Annexes
 - Allocateur de ressources
 - Variantes
 - Réveil multiple
 - Priorité au signalé/signaleur
 - Barrière

29 / 40

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 débloquent 0, 1 ou plusieurs demandeurs selon le nombre de ressources rendues et attendues.



30 / 40

Allocateur – opérations

demander(p)

```

si demande  $\neq$  0 alors -- il y a déjà un demandeur  $\rightarrow$  j'attends mon tour
  Accès.wait
finsi
si  $\neg(\text{nbDispo} \geq p)$  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 moniteurs avec priorité au signaleur \Rightarrow transformer le premier "si" de demander en "tant que" (suffit ici).

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) : $\text{nbDispo} \geq p$
 - libérer(q) : true
- 4 Invariant : $0 \leq \text{nbDispo} \leq N$
- 5 Variable condition : AssezDeRessources



31 / 40

Variante : réveil multiple : signalAll/broadcast

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

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

```

tant que  $\neg(\text{condition d'acceptation})$  faire
  Accès.wait
fintq
...
Accès.signalAll -- battez-vous
    
```

Mauvaise idée ! (performance, prédictibilité)



33 / 40

Réveil multiple : cour de récréation unisexe

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

⑥ $\text{entrer}(\text{genre } g)$ si $\text{nb}(\text{inv}(g)) \neq 0$ alors $\text{accès}(g).\text{wait}$ finsi $\text{nb}(g)++$	$\text{sortir}(\text{genre } g)$ $\text{nb}(g)--$ si $\text{nb}(g) = 0$ alors $\text{accès}(\text{inv}(g)).\text{signalAll}$ finsi
--	--

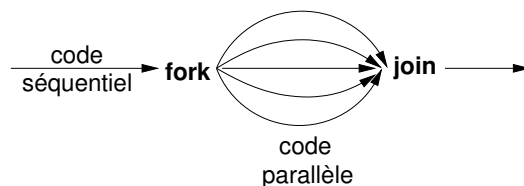
(solution naïve : risque de famine si un genre se coalise pour avoir toujours un membre présent dans la cour)

34 / 40

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 »



36 / 40

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 : évitement de la famine : variable $\text{attente}(\text{genre})$ pour compter les enfants en attente et ne pas accaparer la cour.

```

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

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

35 / 40

Barrière à N activités - méthodologie

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

37 / 40

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)

38 / 40

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
    
```

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 \Rightarrow 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.

39 / 40

Contenu de cette partie

Sixième partie

Programmation multiactivité

Java & Posix Threads

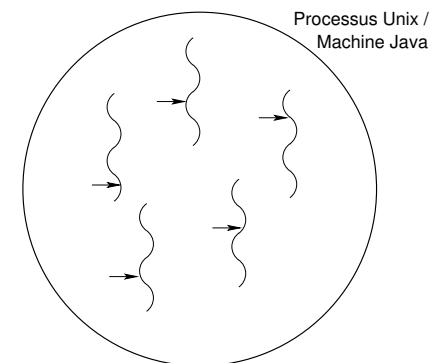
Préparation aux TP : présentation des outils de programmation concurrente autour de la plateforme Java

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

Plan

Processus multiactivité

- 1 Généralités
- 2 Threads Java
 - Manipulation des activités
 - Données localisées
- 3 Synchronisation Java
 - Moniteur Java
 - Autres objets de synchronisation
 - Régulation du parallélisme
 - Synchronisation – java d'origine
- 4 POSIX Threads & autres approches
 - Posix Threads
 - Synchronisation Posix Thread
 - Autres approches



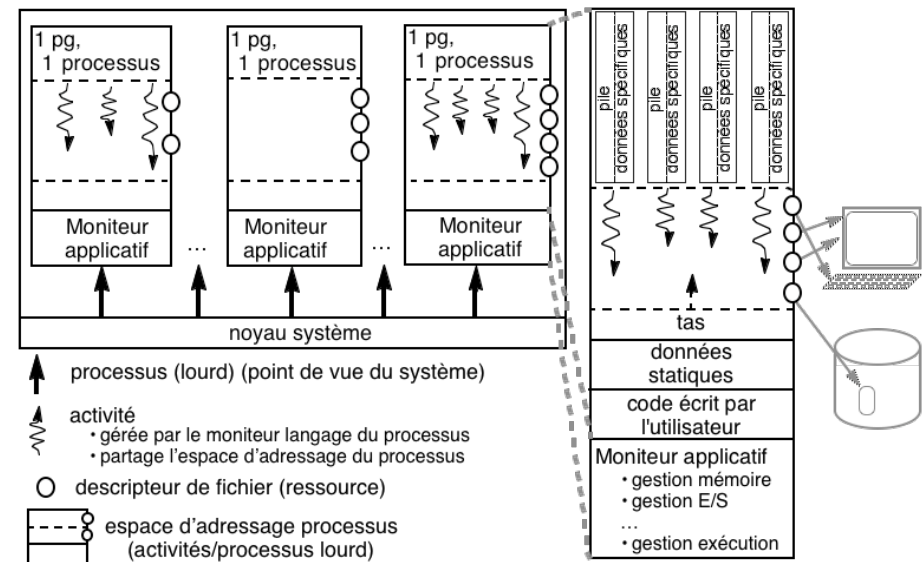
1 espace d'adressage, plusieurs flots de contrôle.
 ⇒ plusieurs **activités** (ou processus légers) au sein d'un même processus UNIX / d'une même machine virtuelle Java.

Processus lourds vs légers

- *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
- *Processus légers* (threads, activités...) :
 - **unité d'exécution** : résulte de la 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
 - chaque activité a sa pile d'exécution et son contexte processeur, mais les autres éléments sont partagés
 - une bibliothèque **applicative** (« moniteur ») gère le partage entre activités du temps processeur alloué au processus lourd
 - commutation plus efficace.

6 / 74

Mise en œuvre des processus légers



7 / 74

Difficultés de mise en œuvre des processus légers

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

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

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

8 / 74

Processeurs virtuels

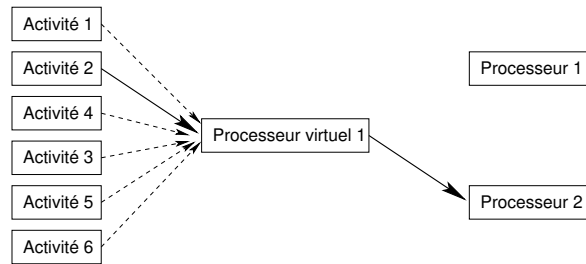
Entre le processeur physique et les activités, il existe généralement une entité interne au noyau, appelé *kernel process* ou *processeur virtuel*.

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

9 / 74

Many-to-one

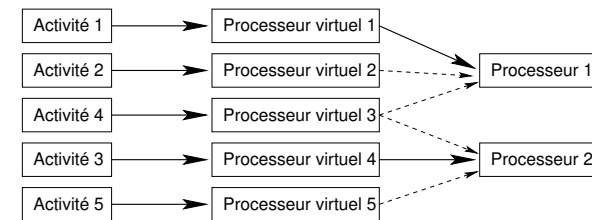


- + commutation entre activités efficace
- + implantation simple et portable
- pas de bénéfice si plusieurs processeurs
- blocage du processus (donc de toutes les activités) en cas d'appel système bloquant, ou implantation complexe



10 / 74

Many-to-many

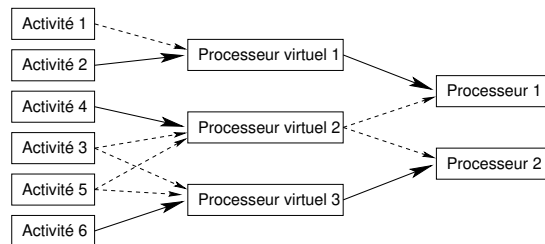


- + vrai parallélisme si plusieurs processeurs physiques
- + pas de blocage des autres activités en cas d'appel bloquant
- commutation moins efficace (dans le noyau)
- ressources consommées élevées



11 / 74

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



12 / 74

Plan

- 1 Généralités
- 2 Threads Java
 - Manipulation des activités
 - Données localisées
- 3 Synchronisation Java
 - Moniteur Java
 - Autres objets de synchronisation
 - Régulation du parallélisme
 - Synchronisation – java d'origine
- 4 POSIX Threads & autres approches
 - Posix Threads
 - Synchronisation Posix Thread
 - Autres approches



13 / 74

Conception d'applications parallèles en Java

Java permet de manipuler

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

Le degré de parallélisme des applications Java peut être

- contrôlé directement (manipulation des threads)
- ou régulé
 - explicitement : interface `java.util.concurrent.Executor`
 - implicitement : programmation asynchrone/fonctionnelle



14 / 74

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

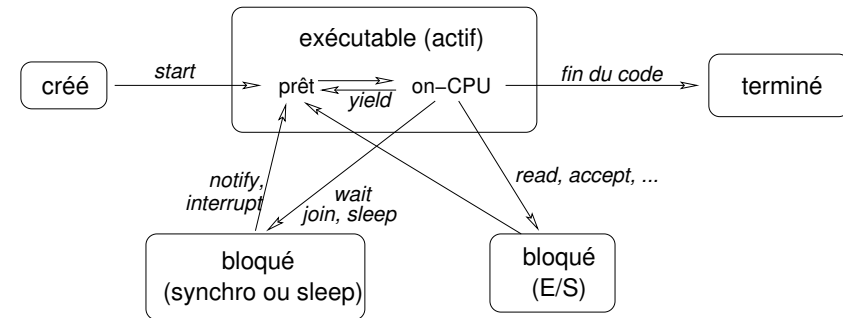
```
Runnable a = new MonActivité(...);
Thread t = new Thread(a); // activité créée
t.start(); // activité démarrée
...
t.join(); // attente de la terminaison
```

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



16 / 74

Cycle de vie d'une activité



15 / 74

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.



18 / 74

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.



20 / 74

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)



19 / 74

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...



21 / 74

Plan

1 Généralités

2 Threads Java

- Manipulation des activités
- Données localisées

3 Synchronisation Java

- Moniteur Java
- Autres objets de synchronisation
- Régulation du parallélisme
- Synchronisation – java d'origine

- Posix Threads
- Synchronisation Posix Thread
- Autres approches

27

22 / 74

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



24 / 74

Objets de synchronisation

Le paquetage `java.util.concurrent` fournit

- une réalisation des moniteurs
- divers autres objets de synchronisation
 - barrière
 - sémaphore
 - compteur
 - ...
- le contrôle du degré de parallélisme : Thread, Executor
- des structures de données autorisant/facilitant les accès concurrents
 - accès atomiques : `ConcurrentHashMap`...
 - accès non bloquants : `ConcurrentLinkedQueue`



23 / 74

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();
    }
}
```



24 / 74

Sémaphores

Sémaphore

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

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

27 / 74

Sémaphores - un producteur/consommateur (2)

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

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

Producteurs/consommateurs

Paquetage `java.util.concurrent`

BlockingQueue

- `BlockingQueue` = producteurs/consommateurs (interface)
- `LinkedBlockingQueue` = prod./cons. à tampon non borné
- `ArrayBlockingQueue` = prod./cons. à tampon borné

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

29 / 74

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.



30 / 74

Atomicité à grain fin

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

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

⇒ synchronisation non bloquante

Danger

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



32 / 74

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)



31 / 74

Plan

- 1 Généralités
- 2 Threads Java
- 3 Synchronisation Java
 - Moniteur Java
 - Autres objets de synchronisation
 - Régulation du parallélisme
 - Synchronisation – java d'origine
- 4 POSIX Threads & autres approches



33 / 74

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



34 / 74

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());
        }
    }
}
```

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.



35 / 74

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];

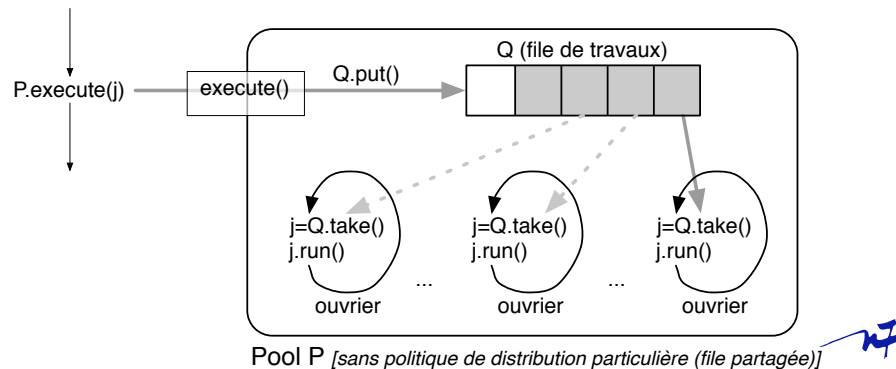
        // lancement des travaux
        for (int i = 0; i < NB; i++) {
            int j = i;
            exec.execute(() -> { System.out.println("hello" + j); });
            res[i] = exec.submit(() -> { return 3 * j; });
        }

        // 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)



38 / 74

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 :

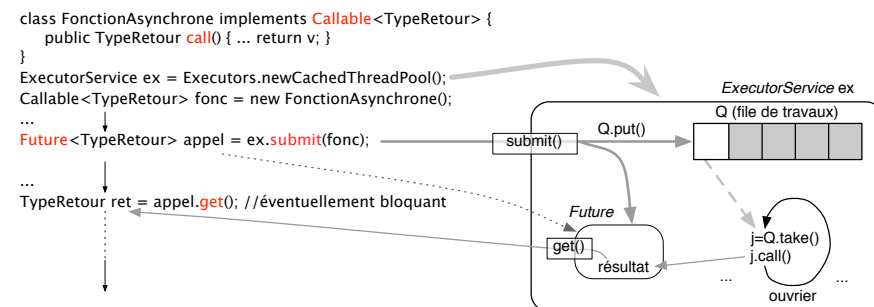
- Nombre fixe d'activités : `newSingleThreadExecutor()`, `newFixedThreadPool(int nThreads)`
- Nombre d'activités adaptable : `newCachedThreadPool()`
 - Quand il n'y a plus d'activité disponible et qu'un travail est déposé, création d'une nouvelle activité
 - Quand la queue est vide et qu'un délai suffisant (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...

40 / 74

Évaluation asynchrone : Callable et Future

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



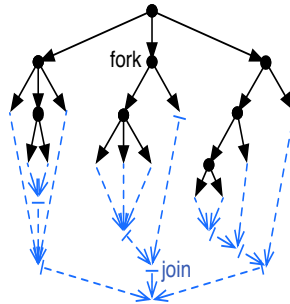
41 / 74

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
  }
}
    
```



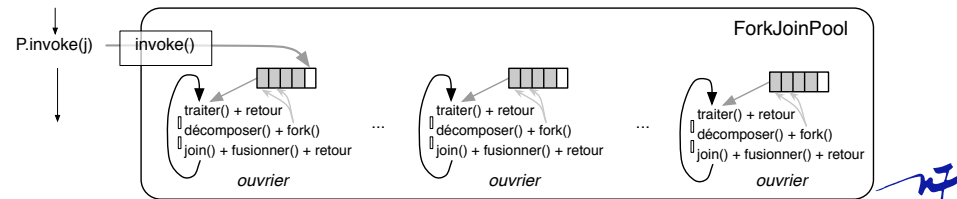
42 / 74

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()`)



43 / 74

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

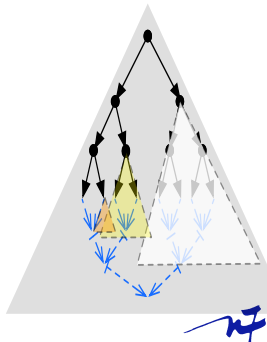
Activité d'un ouvrier du ForkJoinPool :

- 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

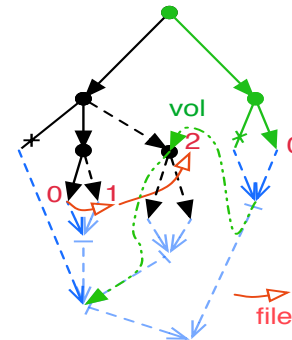
- parcourt et traite en profondeur d'abord → économie d'espace
- construit progressivement en largeur, au fur et à mesure de son parcours : lorsqu'un ouvrier descend d'un niveau, les frères de la tâche à traiter sont créés, et placés en tête de la file d'attente



44 / 74

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.

45 / 74

Plan

- 1 Généralités
- 2 Threads Java
- 3 Synchronisation Java
 - Moniteur Java
 - Autres objets de synchronisation
 - Régulation du parallélisme
 - Synchronisation – java d'origine
- 4 POSIX Threads & autres approches



46 / 74

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)



48 / 74

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



47 / 74

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.



49 / 74


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

demande bloquante	libération
<pre>req = new Requête(...) synchronized(file) { if (satisfiable(req)) { // + maj état applicatif req.ok = true; } else { file.add(req) } } synchronized(req) { while (! req.ok) req.wait(); }</pre>	<pre>synchronized(file) { // + maj état applicatif for (Requête r : file) { synchronized(r) { if (satisfiable(r)) { // + maj état applicatif r.ok = true r.notify(); } } } }</pre>



54 / 74

Posix Threads

Standard de librairie multiactivité pour le C

Contenu de la bibliothèque :

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



56 / 74

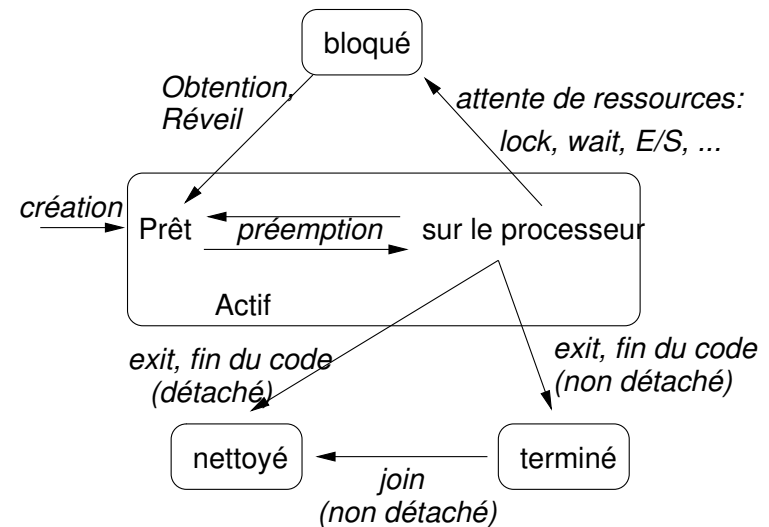
Plan

- 1 Généralités
- 2 Threads Java
 - Manipulation des activités
 - Données localisées
- 3 Synchronisation Java
 - Moniteur Java
 - Autres objets de synchronisation
 - Régulation du parallélisme
 - Synchronisation – java d'origine
- 4 POSIX Threads & autres approches
 - Posix Threads
 - Synchronisation Posix Thread
 - Autres approches



55 / 74

Cycle de vie d'une activité



57 / 74

Création d'une activité

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

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

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

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

58 / 74

L'activité initiale

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

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

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

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

60 / 74

Terminaison

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

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

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

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

```
int pthread_detach (pthread_t thread);
```

Détache l'activité `thread`.
 Les ressources allouées pour l'exécution d'une activité (pile...) ne sont libérées que lorsque l'activité s'est terminée, et que `join` a été effectué *ou* l'activité a été détachée.

59 / 74

Création d'activités - exemple

```
#include <stdio.h>
#include <pthread.h>

struct param { int max; int step; };

void *foo (void *arg) {
    struct param *p = (struct param *)arg;
    for (int i = 0; i < p->max; i += p->step)
        printf ("%d\n", i);
    return NULL;
}

int main() {
    int st;
    pthread_t t1, t2;
    struct param p1 = { 10, 2 };
    struct param p2 = { 15, 3 };
    st = pthread_create(&t1, NULL, foo, &p1);
    if (st != 0) perror("thread creation failed");
    st = pthread_create(&t2, NULL, foo, &p2);
    if (st != 0) perror("thread creation failed");
    pthread_exit (NULL);
}
```

Données spécifiques

Données spécifiques

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

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

int pthread_setspecific (pthread_key_t clef,
                        void *val);

void *pthread_getspecific (pthread_key_t clef);
```



62 / 74

Verrou

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init (pthread_mutex_t *mutex,
                       const pthread_mutex_attr *attr);
int pthread_mutex_destroy (pthread_mutex_t *m);
```

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

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

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

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



64 / 74

Synchronisation PThread

Principe

Moniteur de Hoare élémentaire avec priorité au signaleur :

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

Ordonnancement

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

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



63 / 74

Variable condition

```
pthread_cond_t vc = PTHREAD_COND_INITIALIZER;

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

int pthread_cond_destroy (pthread_cond_t *vc);
```



65 / 74

Attente/signal

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

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

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



66 / 74

Attente/signal

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

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

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



67 / 74

Moniteur PThread : un producteur/consommateur (1)

Un producteur/consommateur d'entiers similaire à celui en Java (transparent 25)

```
#define N 10
int items[N];
int nbElems, depot, retrait ;
pthread_mutex_t lock;
pthread_cond_t pasVide, pasPlein;

void init () {
    pthread_mutex_init(&lock, NULL);
    pthread_cond_init (&pasVide, NULL);
    pthread_cond_init (&pasPlein, NULL);
}
```



68 / 74

Moniteur PThread : un producteur/consommateur (2)

```
void deposer(int val) {
    pthread_mutex_lock(&lock);
    while (nbElems == N) pthread_cond_wait(&pasPlein, &lock);
    items[depot] = val;
    depot = (depot + 1) % N;
    nbElems++;
    pthread_cond_signal (&pasVide);
    pthread_mutex_unlock(&lock);
}

int retirer () {
    pthread_mutex_lock(&lock);
    while (nbElems == 0) pthread_cond_wait(&pasVide, &lock);
    int res = items[retrait];
    retrait = (retrait + 1) % N;
    nbElems--;
    pthread_cond_signal (&pasPlein);
    pthread_mutex_unlock(&lock);
    return res;
}
```

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`

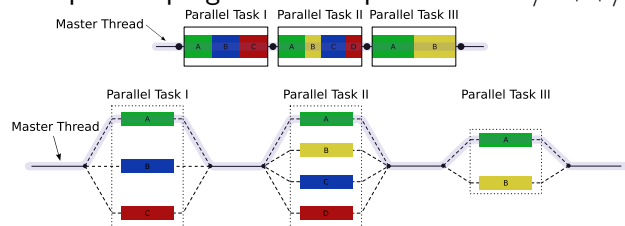
Note : l'API Posix Threads est aussi supportée (ouf).



70 / 74

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;
```



72 / 74

.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();`



71 / 74

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



73 / 74

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)



Contenu de cette partie

Septième partie

Processus communicants

- Modèles de programmation concurrente
- Modèle des processus communicants
- Approche CSP/Go pour la programmation concurrente
 - Goroutine et canaux
 - Communiquer explicitement plutôt que partager implicitement
- Approche Ada pour la programmation concurrente
 - Tâches et rendez vous
 - Démarche de conception d'applications concurrentes en Ada
 - Transposition de la démarche vue dans le cadre de la mémoire partagée (moniteurs)
 - Extension tirant parti des possibilités de contrôle fin offertes par Ada

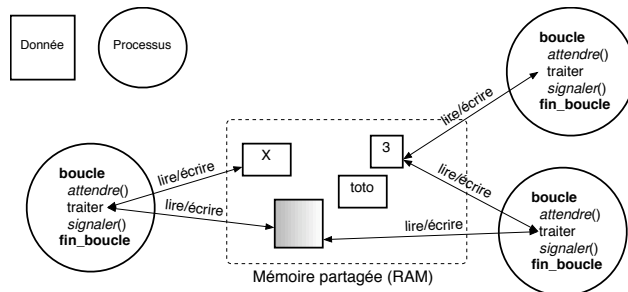


2 / 60



3 / 60

Modèles d'interaction : mémoire partagée

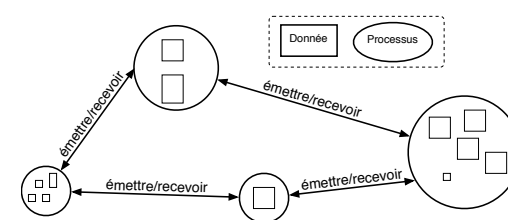


- **Données partagées**
- Communication implicite
 - résulte de l'accès et de la manipulation des variables partagées
 - l'identité des activités n'intervient pas dans l'interaction
- **Synchronisation explicite** (et nécessaire)
- Architectures/modèles cibles
 - multiprocesseurs à mémoire partagée,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



4 / 60

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



5 / 60

Plan

- 1 Processus communicants
 - Principes
 - Désignation, alternatives
 - Architecture d'une application parallèle
- 2 Communication synchrone – CSP/CCS/Go
 - Principes
 - Recherche concurrente
 - Exemples d'objets de synchronisation
- 3 Rendez-vous étendu – Ada
 - Principe du rendez-vous
 - Mise en œuvre en Ada
 - Méthodologie par machine à états



6 / 60

Processus communicants



Principes

- Communication inter-processus avec des **opérations explicites d'envoi / réception** de messages
- Synchronisation via ces primitives de communication **bloquantes** : envoi (bloquant) de messages / réception bloquante de messages
- Communicating Sequential Processes (CSP) / Calculus of Communicating Systems (CCS) / π -calcul / Erlang / Go
- Ada

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



7 / 60

Quelle synchronisation ?



Réception

Réception bloquante : attendre un message

Émission

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

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



8 / 60

Désignation du destinataire et de l'émetteur



Nommage

- Direct : désignation de l'activité émettrice/destinataire
SEND message TO processName
RECV message FROM processName
- Indirect : désignation d'une boîte à lettres ou d'un **canal de communication**
SEND message TO channel
RECV message FROM channel



9 / 60

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)



10 / 60

Divers

Émission asynchrone \Rightarrow risque de buffers pleins

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

Émission non bloquante \rightarrow émission bloquante

introduire un acquittement

(SEND m TO ch; RECV _ FROM ack)

|| (RCV m FROM ch; SEND _ TO ack)

Émission bloquante \rightarrow émission non bloquante

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

(SEND m TO ch1)

|| boucle (RCV m FROM ch1; insérer m dans file)

|| boucle (si file non vide alors extraire et SEND TO ch2)

|| (RCV FROM ch2)



12 / 60

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)

(RCV msg1 FROM channel1) OR (SEND msg2 TO channel2)

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

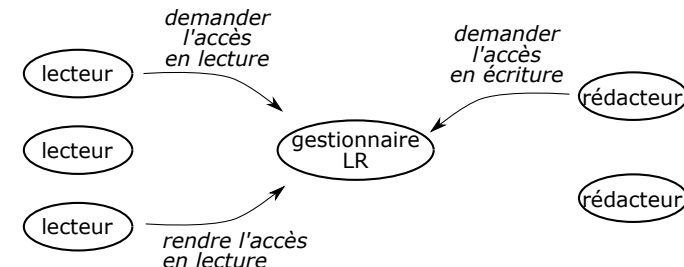


11 / 60

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**.



13 / 60

Activité arbitre pour un objet partagé



Interactions avec l'objet partagé

Pour chaque opération *Op*,

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

(⇒ se synchroniser avec l'arbitre)

Schéma de fonctionnement de l'arbitre

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

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



14 / 60

Plan

- 1 Processus communicants
 - Principes
 - Désignation, alternatives
 - Architecture d'une application parallèle
- 2 Communication synchrone – CSP/CCS/Go
 - Principes
 - Recherche concurrente
 - Exemples d'objets de synchronisation
- 3 Rendez-vous étendu – Ada
 - Principe du rendez-vous
 - Mise en œuvre en Ada
 - Méthodologie par machine à états



16 / 60

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



15 / 60

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 indépendamment (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



17 / 60

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)



18 / 60

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.")
}
```



20 / 60

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")
}
```



19 / 60

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>



21 / 60

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
}

```



22 / 60

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

```



24 / 60

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
}

```



23 / 60

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
}

```



25 / 60

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

```

26 / 60

Bilan

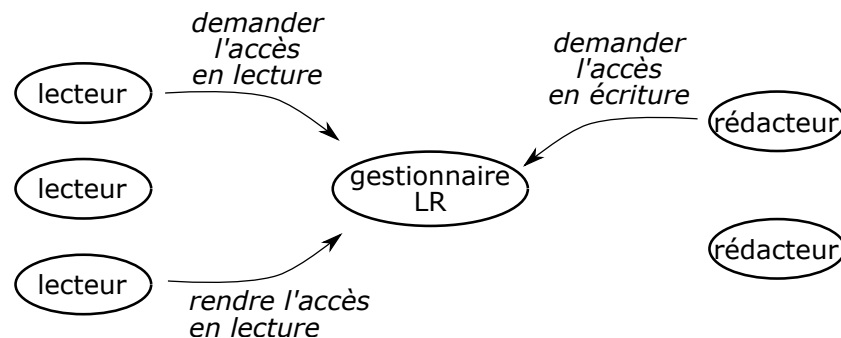
- Création ultra-légère de goroutine : penser concurrent
- Pas besoin de variables partagées
⇒ Pas de verrous
- Pas besoin de variable condition / sémaphore pour synchroniser
- Pas besoin de callback ou d'interruption

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

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

27 / 60

Lecteurs/rédacteurs



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

28 / 60

Lecteurs/rédacteurs



Utilisateur

```

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

```

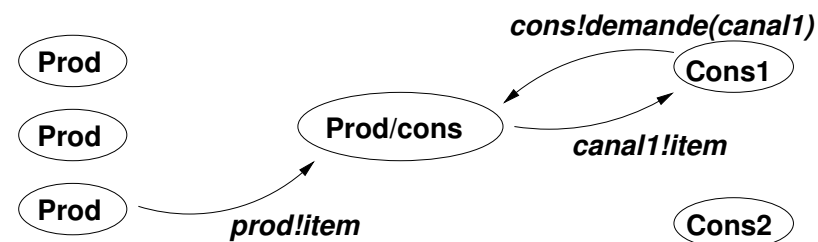
29 / 60

Goroutine de synchronisation

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

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

Producteurs/consommateurs : architecture



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

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

Producteurs/consommateurs



Programme principal

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

Producteurs/consommateurs



Producteur

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

Consommateur

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

Producteurs/consommateurs



Goroutine de synchronisation

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

34 / 60

Modèle Ada

Intérêt

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

Vocabulaire : tâche = activité

36 / 60

Plan

- 1 Processus communicants
 - Principes
 - Désignation, alternatives
 - Architecture d'une application parallèle
- 2 Communication synchrone – CSP/CCS/Go
 - Principes
 - Recherche concurrente
 - Exemples d'objets de synchronisation
- 3 Rendez-vous étendu – Ada
 - Principe du rendez-vous
 - Mise en œuvre en Ada
 - Méthodologie par machine à états

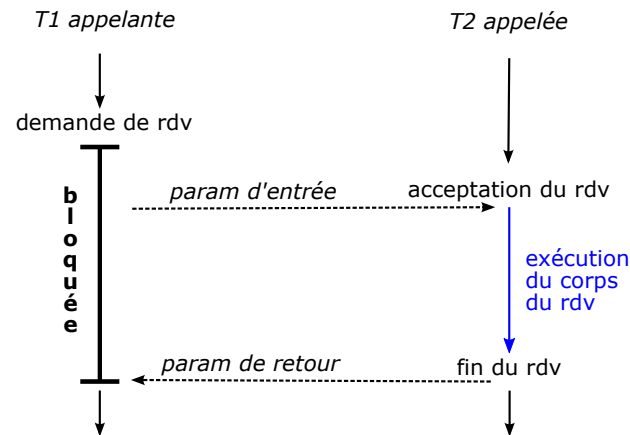
35 / 60

Principe du rendez-vous

- Une tâche possède des **points d'entrée de rendez-vous**.
- Une tâche peut :
 - demander un rendez-vous avec une autre tâche désignée explicitement;
 - attendre un rendez-vous sur un (ou plusieurs) point(s) d'entrée.
- Un rendez-vous est **dissymétrique** : tâche appelante ou cliente vs tâche appelée ou serveur.
- Échanges de données :
 - lors du début du rendez-vous, de l'appelant vers l'appelé;
 - lors de la fin du rendez-vous, de l'appelé vers l'appelant.

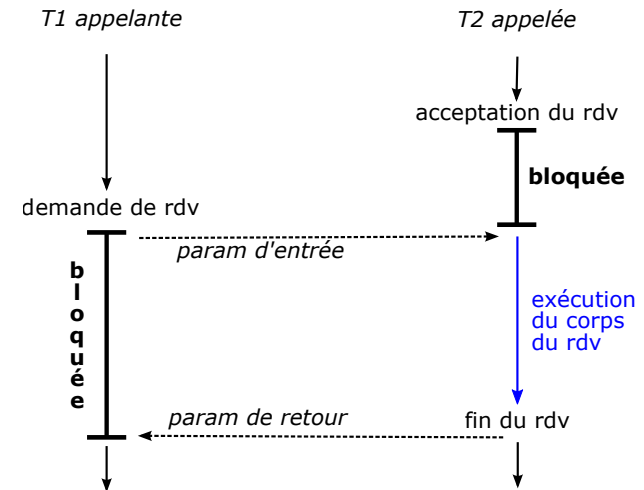
37 / 60

Rendez-vous – client en premier



38 / 60

Rendez-vous – serveur en premier



39 / 60

Principe du rendez-vous

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

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



40 / 60

Déclaration d'une tâche



Déclaration

```

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

Exemple

```

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



41 / 60

Appel de rendez-vous



Appel de rendez-vous

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

Syntaxe identique à un appel de procédure, sémantique bloquante.

Exemple

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



42 / 60

Acceptation d'un rendez-vous



Acceptation

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

Exemple

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



43 / 60

Acceptation parmi un ensemble



Alternative gardée

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



44 / 60

Producteurs/consommateurs



Déclaration du serveur

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

Client : utilisation

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



45 / 60

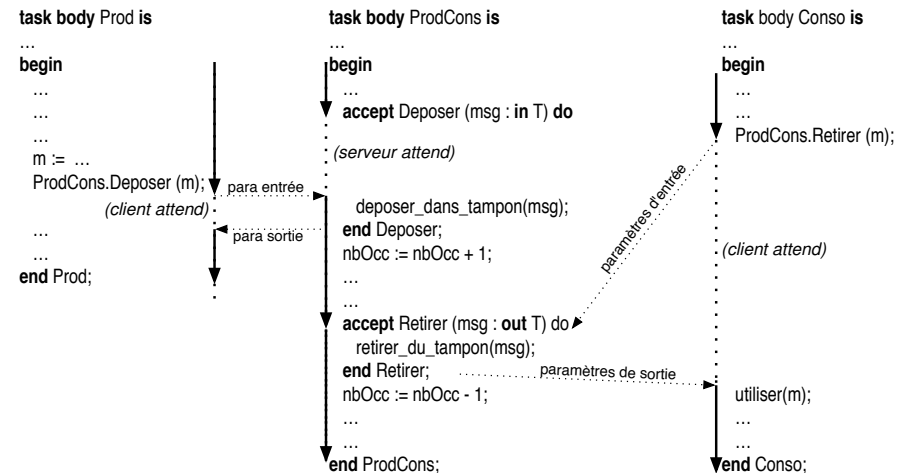

```

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

```

46 / 60

Producteurs/consommateurs – un exemple d'exécution



47 / 60

Remarques

- Les accept ne peuvent figurer que dans le corps des tâches.
- accept sans corps → synchronisation pure.
- Une file d'attente (FIFO) est associée à chaque entrée.
- rdv'count (attribut des entrées) donne le nombre de clients en attente sur une entrée donnée.
- La gestion et la prise en compte des appels diffèrent par rapport aux moniteurs :
 - la prise en compte d'un appel au service est déterminée par le serveur ;
 - plusieurs appels à un même service peuvent déclencher des traitements différents ;
 - le serveur peut être bloqué, tandis que des clients attendent.

48 / 60

Allocateur de ressources

Déclaration du serveur

```

task Allocateur is
  entry Demander (nbDemandé : in natural;
                  id : out array of Ressourceld);
  entry Rendre (nbRendu : in natural;
                id : in array of Ressourceld);
end Allocateur;

```

49 / 60

```

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

```



50 / 60

```

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

```

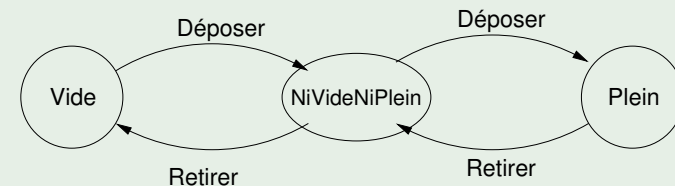
Méthodologie par machine à états



Construire un automate fini à états :

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

Producteurs/consommateurs à 2 cases



51 / 60

```

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

```



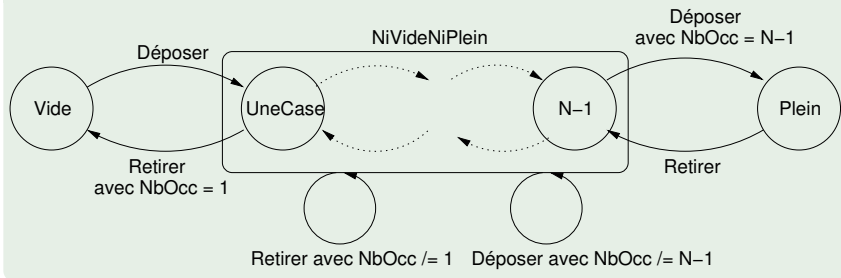
53 / 60

Automate paramétré



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

Producteurs/consommateurs à N cases

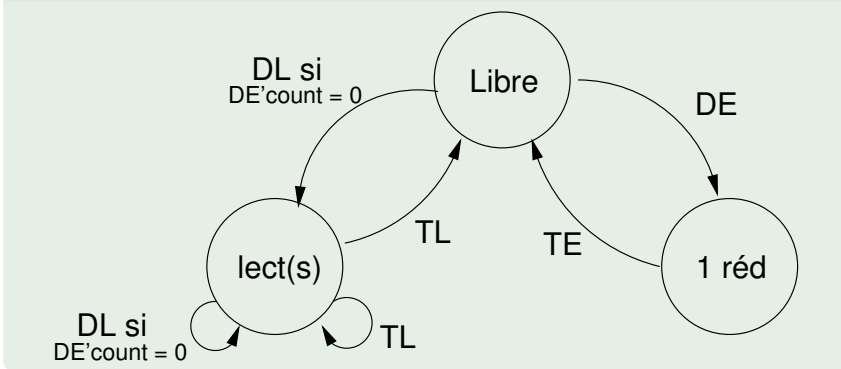


54 / 60

Lecteurs/rédacteurs priorité rédacteurs



Lecteurs/rédacteurs priorité rédacteurs

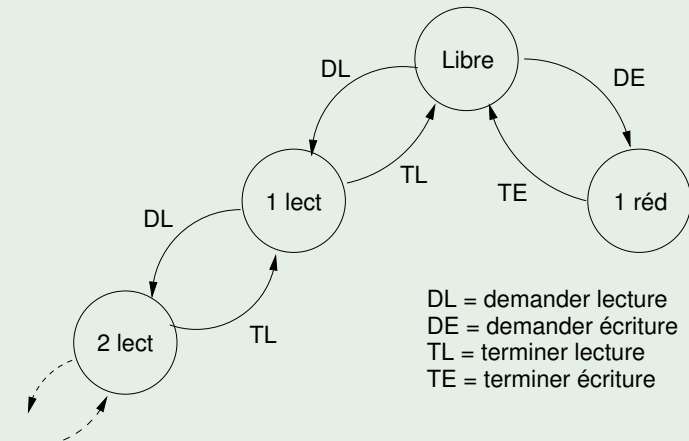


56 / 60

Lecteurs/rédacteurs



Lecteurs/rédacteurs



55 / 60

task body LRprioRed is

type EtatT is (Libre, Lect, Red);

etat : EtatT := Libre;

nblect : Natural := 0;

begin

loop

if etat = Libre then

select

when DE'count = 0 => accept DE; etat := Red;

or

accept DL; etat := Lect; nblect := 1;

end select;

elsif etat = Lect then

select

when DE'count = 0 => accept DE; etat := Red;

or

accept TL; nblect := nblect - 1;

if nblect = 0 then etat := Libre; else etat := Lect; end if;

end select;

elsif etat = Red then

accept TE;

etat := Libre;

end if;

end loop;

end LRprioRed;

Dynamicité : activation de tâche

Une tâche peut être activée :

- statiquement : chaque **task** T , déclarée explicitement, est activée au démarrage du programme, avant l'initialisation des modules qui utilisent $T.entry$.
- dynamiquement :
 - déclaration par **task type** T
 - activation par allocation : `var t is access T := new T;`
 - possibilité d'activer plusieurs tâches d'interface T .



58 / 60

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)



60 / 60

Dynamicité : Terminaison

Une tâche T est potentiellement appelante de T' si

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

Une tâche se termine quand :

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

La terminaison est difficile !



59 / 60

Contenu de cette partie

Huitième partie

Transactions

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



2 / 59



3 / 59

Plan

- 1 **Transaction**
 - Interférences entre actions
 - Définition des transactions
- 2 Atomicité/Tout ou rien
- 3 Contrôle de concurrence
 - Principe
 - Modélisation
 - Méthodes
 - Cohérence affaiblie
- 4 Mémoire transactionnelle
 - Intégration dans un langage
 - Difficultés
 - Implantation : STM, HTM



4 / 59

Interférences et isolation

Objets partagés + actions concurrentes \Rightarrow résultats cohérents ?

Approches :

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



5 / 59

Contraintes d'intégrité

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

Exemple

Base de données bancaire

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

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

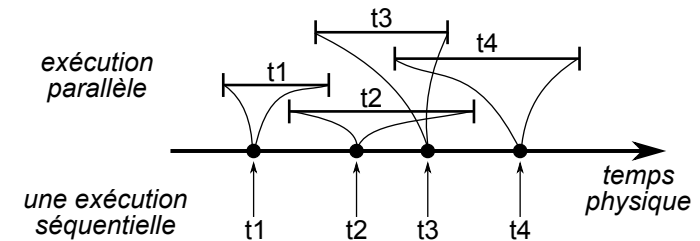
6 / 59

Transaction

Définition

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

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



7 / 59

Transaction - exemple

Invariant $x + y = z$

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

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

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

8 / 59

Transaction - exemple

Invariant $x = y$

T1	T2
1. $x \leftarrow x + 100$	$\alpha.$ $x \leftarrow x * 2$
2. $y \leftarrow y + 100$	$\beta.$ $y \leftarrow y * 2$

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

T1	T2
1. $x \leftarrow x + 100$	$\alpha.$ $x \leftarrow x * 2$
2. $y \leftarrow y + 100$	$\beta.$ $y \leftarrow x$

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

9 / 59

Propriétés ACID

Propriétés ACID

- Atomicité** ou « tout ou rien » : en cas d'abandon (volontaire ou subi), aucun effet visible
- Cohérence** : respect des contraintes d'intégrité
- Isolation** : pas d'interférences entre transactions = pas d'états intermédiaires observables
- Durabilité** : permanence des effets d'une transaction validée



10 / 59

Service transactionnel

Interface du service :

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



12 / 59

Annulation/abandon

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



11 / 59

Comment évaluer la cohérence *efficacement* ?

Objectif

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

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



13 / 59

Mise en œuvre

Propagation des valeurs écrites

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

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

Contrôle de concurrence

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

→ Cohérence et isolation, comme si chaque transaction était seule

Ces deux politiques se combinent \pm bien.



14 / 59

Atomicité (tout ou rien)

Objectif

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

Difficulté

Tenir compte de la possibilité de pannes en cours

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



16 / 59

Plan

- 1 Transaction
 - Interférences entre actions
 - Définition des transactions
- 2 Atomicité/Tout ou rien
- 3 Contrôle de concurrence
 - Principe
 - Modélisation
 - Méthodes
 - Cohérence affaiblie
- 4 Mémoire transactionnelle
 - Intégration dans un langage
 - Difficultés
 - Implantation : STM, HTM



15 / 59

Abandon sans effet

Comment abandonner une transaction sans effet ?

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



17 / 59

Mise en œuvre de l'atomicité

Opérations de base

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

Réalisation de *défaire* et *refaire*

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

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



18 / 59

Approche optimiste : propagation en continu

Utilisation d'un journal des valeurs avant

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

Problèmes liés aux abandons

- Rejets en cascade

```
(1) écrire(x,10) || (2) lire(x)
                  || (3) écrire(y, 8)
(4) abandon()    || → abandonner aussi
```

- Perte de l'état initial

initialement : x=5
 (1) écrire(x,10) || (2) écrire(x,8) -- sauve "x valait 10"
 (3) abandon() || (4) abandon() → x=10 au lieu de x=5



20 / 59

Approche pessimiste : propagation différée

Utilisation d'un journal des valeurs après

Principe

- Écriture dans un espace de travail privé, en mémoire volatile
→ adapté aux mécanismes de gestion mémoire (caches...)
- Journalisation de la validation
- écrire → préécriture dans l'espace de travail
- valider → recopier l'espace de travail en mémoire stable (*liste d'intentions*), puis copier celle-ci en mémoire permanente
→ protection contre les pannes en cours de validation
- défaire → libérer l'espace de travail
- refaire → reprendre la recopie de la liste d'intentions



19 / 59

Plan

- 1 Transaction
 - Interférences entre actions
 - Définition des transactions
- 2 Atomicité/Tout ou rien
- 3 Contrôle de concurrence
 - Principe
 - Modélisation
 - Méthodes
 - Cohérence affaiblie
- 4 Mémoire transactionnelle
 - Intégration dans un langage
 - Difficultés
 - Implantation : STM, HTM



21 / 59

Contrôle de concurrence

Objectif

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

- 1 Exécution **sérialisée** : isolation par exclusion mutuelle.
- 2 Exécution **sérialisable** : contrôler l'entrelacement des actions pour que *l'effet final* soit équivalent à une exécution sérialisée.
 - Il peut exister plusieurs exécutions sérialisées équivalentes
 - Contrôle automatique



22 / 59

Conflits

Idée

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

Conflit

Opérations non commutatives exécutées sur un même objet



24 / 59

Sémantique

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



23 / 59

Exemples de conflits

Exemple

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

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

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

Exemple

	lire	écrire	incrémenter	décrémenter
lire	OK	—	—	—
écrire	—	—	—	—
incrémenter	—	—	OK	OK
décrémenter	—	—	OK	OK



25 / 59

Graphe de dépendance, sérialisabilité

Définition

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

Définition

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

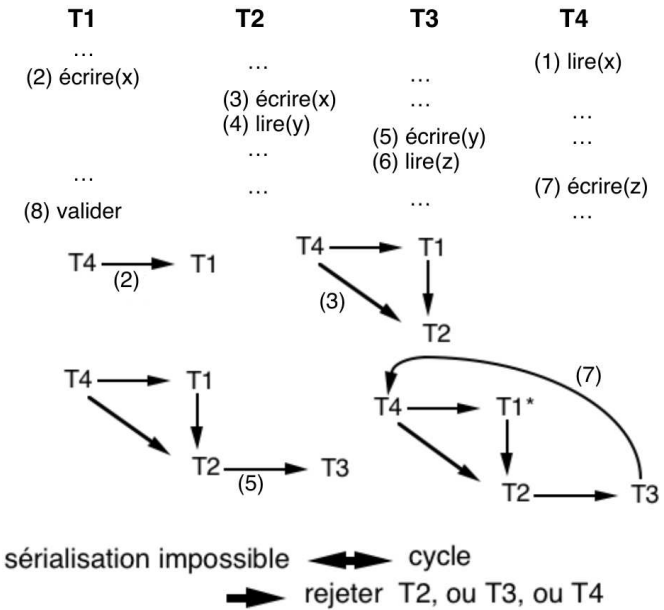
Théorème

Exécution sérialisable \Leftrightarrow son graphe de dépendance est acyclique.



26 / 59

Exemple



27 / 59

Contrôle de concurrence

Quand vérifier la sérialisabilité :

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

Comment garantir la sérialisabilité :

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



28 / 59

Certification (concurrence explicite)

Algorithme

```
-- écritures en mémoire privée avec recopie à la validation
T.lus, T.écrits : objets lus/écrits par T
T.concur : transactions ayant validé pendant l'exéc. de T
actives : ensemble des transactions en cours d'exécution

procédure Certifier(T) :
si (  $\forall T' \in T.concur : T.lus \cap T'.écrits = \emptyset$  )
alors
    -- T peut valider
     $\forall T' \in actives : T'.concur \leftarrow T'.concur \cup \{T\}$ 
sinon
    -- abandon de T
fin
```

Protocole coûteux et coût du rejet \Rightarrow faible taux de conflit



29 / 59

Certification (estampille)

Algorithme

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

procédure Certifier(T) :
si ( $\forall T' : T.déb < T'.val < T.fin : T.lus \cap T'.écrits = \emptyset$ )
alors
    C ← C + 1
    T.val ← C
sinon
    abandon de T
fin
```

Protocole coûteux et coût du rejet \Rightarrow faible taux de conflit



30 / 59

Estampilles : abandon par prudence

Abandon superflu

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

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



32 / 59

Contrôle continu par estampilles

Ordre de sérialisation = ordre des estampilles

Algorithme

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

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

procédure écrire(T,O,v)
si T.E  $\geq$  O.lect  $\wedge$  T.E  $\geq$  O.réd
alors
    écriture de O possible
    O.red ← T.E
sinon
    abandon de T
finsi
```

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



31 / 59

Estampilles : amélioration

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

Algorithme

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



33 / 59

Contrôle continu par verrouillage à deux phases

Ordre de sérialisation = ordre chronologique d'accès aux objets

$T_1 \rightarrow T_2$: **bloquer** T_2 jusqu'à ce que T_1 valide.

Verrous en lecture/écriture/...

Si toute transaction est

- bien formée (prise du verrou avant une opération)
- à deux phases (pas de prise de verrou après une libération)
 - phase 1 : acquisitions et opérations
 - point de validation
 - phase 2 : libérations

alors la sérialisation est assurée.

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

nf

34 / 59

Verrouillage à deux phases : justification du protocole

Idée de base

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

Schéma de preuve

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

nf

36 / 59

Nécessité des deux phases

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

Invariant $x = y$
 initialement $x = y = 4$

T_1	T_2
1. <i>lock</i> x	$\alpha.$ <i>lock</i> x
2. $x \leftarrow x + 1$	$\beta.$ <i>lock</i> y
3. <i>unlock</i> x	$\gamma.$ $x \leftarrow x * 2$
4. <i>lock</i> y	$\delta.$ $y \leftarrow y * 2$
5. $y \leftarrow y + 1$	$\epsilon.$ <i>unlock</i> y
6. <i>unlock</i> y	$\zeta.$ <i>unlock</i> x

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

nf

35 / 59

Verrouillage à deux phases : interblocage

Que faire en cas de conflit de verrouillage :

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

nf

37 / 59

Verrouillage strict à deux phases

- Prise implicite du verrou au premier accès à une variable
- Libération automatique à la validation/abandon
- Garantit simplement les deux phases
- Tout se fait à la validation : simple
- Restriction du parallélisme
(conservation des verrous jusqu'à la fin)



38 / 59

Moins que sérialisable ?

La sérialisabilité est parfois un critère trop fort → **cohérence faible**.

SQL définit quatre niveaux d'isolation :

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



40 / 59

Comment garantir la cohérence *efficacement* ?

Objectif

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

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

Gestionnaire de contention

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



39 / 59

Incohérences tolérables ?

Pertes de mises à jour

Écritures écrasées par d'autres écritures.

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

Lectures sales

Écritures abandonnées mais observées

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



41 / 59

Incohérences tolérables ?

Lectures non répétables

Donnée qui change de valeur pendant la transaction

```
(1) a := lire(x);
(2) b := lire(x);
```

```
(a) écrire(x,100);
```

Lectures fantômes

Donnée agrégée qui change de contenu

<pre> (0) sum := 0; (1) nb := cardinal(S) </pre>	<div style="border-left: 1px solid black; height: 100px; margin: 0 5px;"></div>	<pre> (a) ajouter(S, 15); </pre>
<pre> (2) $\forall x \in S : \text{sum} := \text{sum} + x$ (3) moyenne := sum / nb </pre>		

42 / 59

Plan

1 Transaction

- Interférences entre actions
- Définition des transactions

2 Atomicité/Tout ou rien

- Principe
- Modélisation
- Méthodes
- Cohérence affaiblie

4 Mémoire transactionnelle

- Intégration dans un langage
- Difficultés
- Implantation : STM, HTM

44 / 59

Conclusion

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

43 / 59

Mémoire transactionnelle

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

45 / 59

Intégration explicite dans un langage

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

Interface exposée

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



46 / 59

Transaction et synchronisation

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

Intégrer la synchronisation dans les transactions

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

retry

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



48 / 59

Intégration dans un langage

Introduire un bloc atomique :

atomically

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

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



47 / 59

Transaction et synchronisation 2

Exécuter une autre transaction si la première échoue :

orelse

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



49 / 59

Cohérence interne

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

init x=y	
atomic {	atomic {
if (x != y)	x++;
while (true) {}	y++;
}	}
int *x; bool nonnul;	
atomic {	atomic {
if (nonnul)	x ← NULL;
*x ← 3;	nonnul ← false;
}	}

Transaction zombie (ou condamnée) mais visible.



50 / 59

Actions non annulables

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

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



52 / 59

Interaction avec code non transactionnel

Lectures non répétables

Donnée qui change de valeur

atomic {		écriture(x,100);
a := lire(x);		
b := lire(x);		
}		

Lectures sales

Écritures abandonnées mais observées

atomic {		b := lire(x);
écriture(x,100);		
abandon;		
}		



51 / 59

Transaction et exception

Exception dans une transaction ?

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

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



53 / 59

Imbrication

Transaction imbriquée

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

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

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

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



54 / 59

HTM – Hardware Transactional Memory

Instructions processeur

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

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

Implantation

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



56 / 59

STM – Software Transactional Memory

Implantation purement logicielle de la mémoire transactionnelle.

Interface explicite

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

Programmation explicite, ou insertion par le compilateur.

Points critiques

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

Nombreuses implantations, beaucoup de variété.



55 / 59

HTM – limitations

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



57 / 59

Implantation hybride

Coopération STM/HTM

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

Implantation STM sur HTM

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

Implantation STM avec assistance matérielle

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

Conclusion

Programmation concurrente par transactions :

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

Neuvième partie

Synchronisation non bloquante



2 / 32

Limitation des verrous



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

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



4 / 32

Plan

1 Objectifs et principes

2 Exemples

- Splitter & renommage
- Pile chaînée
- Liste chaînée

3 Conclusion



3 / 32

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)



5 / 32

Synchronisation non bloquante



Non-blocking synchronization

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

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

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



6 / 32

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)



8 / 32

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)



7 / 32

Plan

1 Objectifs et principes

2 Exemples

- Splitter & renommage
- Pile chaînée
- Liste chaînée

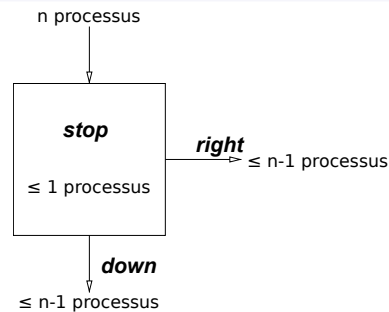
3 Conclusion



9 / 32

Splitter

Moir, Anderson 1995



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

10 / 32

Schéma de preuve

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

Vivacité ni boucle ni blocage

stop si $x = 1$ évident (une seule activité exécute *direction()*)

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

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

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

12 / 32

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;
  
```

11 / 32

Renommage



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

Solution à base de verrous

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

Solution non bloquante

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

13 / 32

Grille de splitters

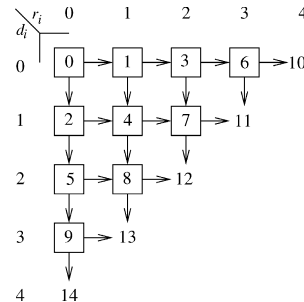


Étiquettes uniques : un splitter renvoie **stop** à une activité au plus

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

Toute activité obtient une étiquette :

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



14 / 32

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

15 / 32

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

16 / 32

Synchronisation classique



Conflit push/push, pop/pop, push/pop \Rightarrow 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

18 / 32

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

19 / 32

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; }
        }
    }
}
```

20 / 32

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

22 / 32

Synchronisation classique

Conflit enfile/enfiler, retirer/retirer, enfile/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 enfile/défile

23 / 32

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; }
    }
}
```

24 / 32

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
```

25 / 32

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;
```

27 / 32

Problème A-B-A

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



29 / 32

Plan

- 1 Objectifs et principes
- 2 Exemples
 - Splitter & renommage
 - Pile chaînée
 - Liste chaînée
- 3 Conclusion



31 / 32

Solutions au problème A-B-A

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



30 / 32

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`



32 / 32

Dixième partie

Conclusion

Conclusion

Programmation parallèle

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

Deux aspects

- le parallélisme
- la synchronisation



2 / 6



3 / 6

Approches

Approches traditionnelles

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

Exemples : Java Thread, C POSIX Threads



4 / 6

Approches

Approches modernes

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

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



5 / 6

Approches d'avenir (?)

- Création implicite et explicite d'activités
- Synchronisation implicite
 - mémoire transactionnelle
 - bibliothèque de structures de données non bloquantes
- Absence d'effets de bord :
 - langages fonctionnels (Haskell)
 - parallélisation paresseuse
 - programmation événementielle ou dataflow

