

Histoires de temps en informatique

Concurrence, parallélisme, synchronisation

Sacha Krakowiak

Université de Grenoble & Aconit

Le temps

❖ Le temps, une notion familière ?

« Qu'est-ce donc que le temps ? Si personne ne m'interroge, je le sais ; si je veux répondre à cette demande, je l'ignore »

saint Augustin

❖ Le temps en informatique, une notion multiforme

temps physique ou temps logique(s) ?

temps continu ou temps discret ?

temps linéaire ou temps ramifié ?

❖ Histoire de 3 aspects liés au temps

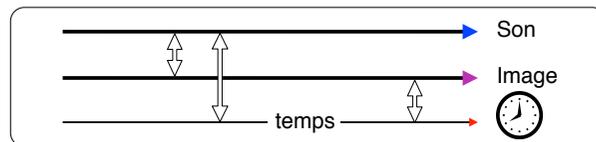
concurrence

parallélisme

synchronisation

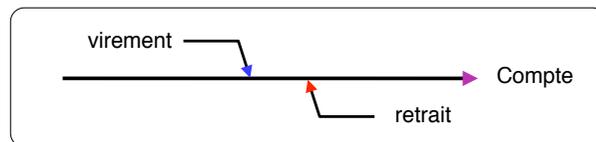
Synchronisation

Vidéo



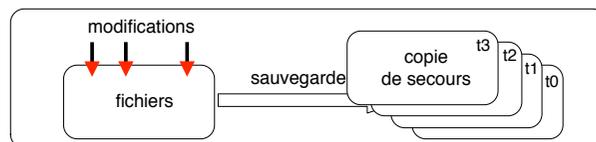
alignement strict

Banque



ordre de précedence

Time Machine



événements datés

❖ Activité

séquence linéaire d'actions, dont chacune modifie l'état du système

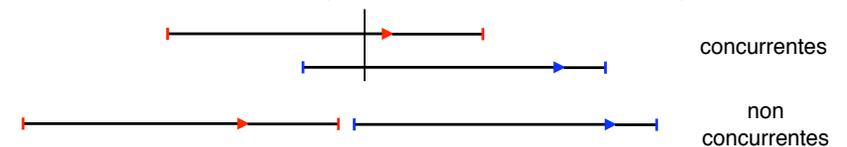
l'organe qui exécute les actions est un *processeur*
(pas nécessairement celui d'un ordinateur)

l'organe qui mesure la progression est une *horloge*
(pas nécessairement physique)

une activité a un début et (éventuellement) une fin

❖ Concurrence

deux activités sont *concurrentes* si (à un point de leur progression) elles sont en cours (commencées et non terminées)



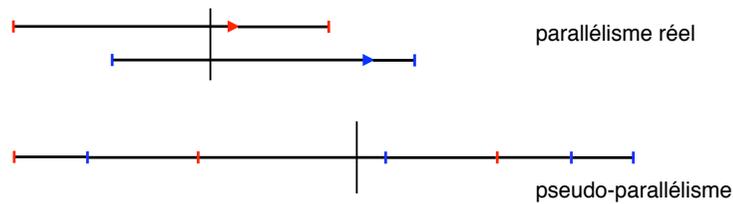
Parallélisme

❖ Activités parallèles

deux activités sont *parallèles* si elles sont concurrentes *et* s'exécutent sur des processeurs différents
les horloges peuvent être ou non synchrones

❖ Parallélisme et pseudo-parallélisme

deux activités peuvent être concurrentes sans être parallèles (s'il n'y a qu'un seul processeur)

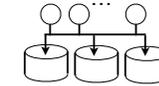


Le temps en informatique

❖ Performances du matériel

	Temps d'accès à la mémoire	Fréquence d'horloge
1950	200 μ s	0,5 KHz
2014	10 ns ($2 \cdot 10^4$)	10 GHz ($2 \cdot 10^7$)

❖ Temps de réponse d'une requête



0,45 s

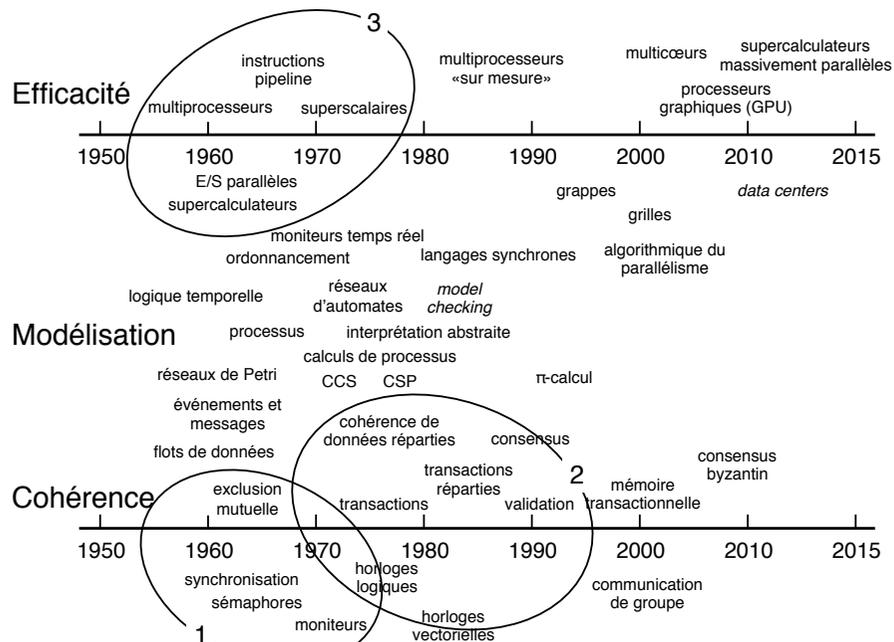
❖ Divers aspects du temps

mesure d'efficacité (ci-dessus)

garanties et preuve de propriétés

outils pour la cohérence et la synchronisation (exemples antérieurs)

base pour la modélisation (logique temporelle, etc.)



Les débuts de la synchronisation

Les débuts de la programmation concurrente

❖ Début des années 1960...

premiers systèmes d'exploitation multiprogrammés
 premières entrées-sorties en parallèle
 premiers multiprocesseurs

❖ La concurrence : une notion très mal comprise...

pas de modèle conceptuel
 des instruments mal maîtrisés

matériels : les interruptions (1955-56)
 logiciels : les instructions *fork-join* (1963)

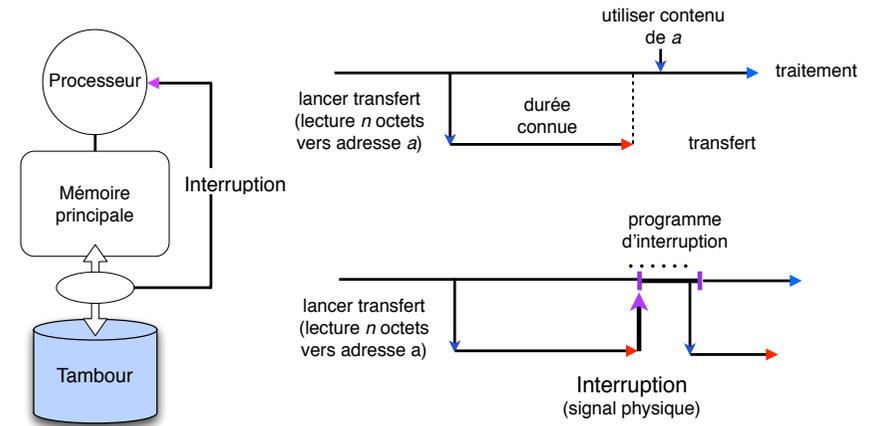
explications plus loin

❖ ... et des conséquences désastreuses

comportement non reproductible
 pannes inexplicables

Programmer la concurrence (1)

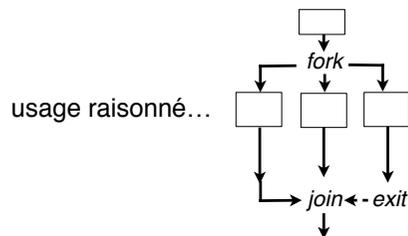
❖ Entrées-sorties asynchrones



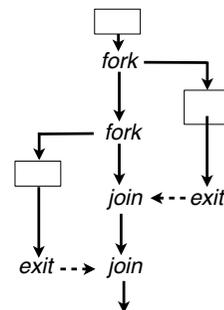
Programmer la concurrence (2)

❖ Programmer un multiprocesseur

premières idées (1963) : instructions *fork* et *join*



... mais dévoiement possible
 (analogue au *go to*)

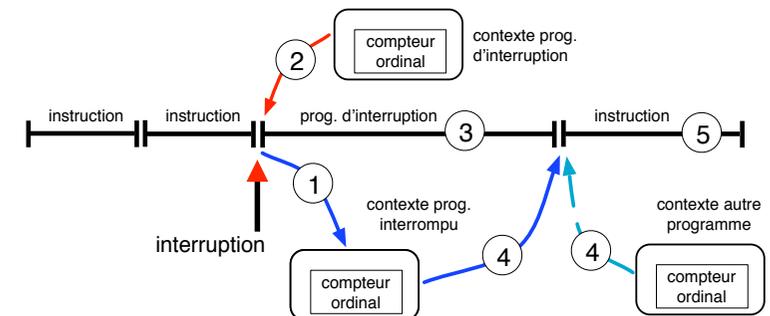


Mécanisme de l'interruption

L'origine du signal est extérieure au processeur interrompu (horloge, fin d'entrée-sortie, capteur externe, etc.). Des classes («niveaux») d'interruption distinguent les différentes causes.

Un niveau peut être «masqué» (l'interruption sera ignorée)

L'interruption est prise en compte en un «point interruptible» (en général entre deux instructions)

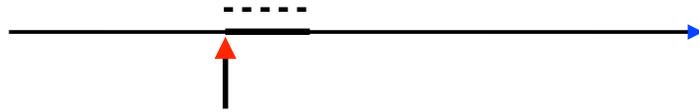


Dangers des interruptions (1)

«C'était une grande invention, mais aussi une boîte de Pandore.»

Edsger Dijkstra

❖ Effet d'une interruption (en l'absence de précautions)



insérer une séquence d'instructions «n'importe où» dans le corps d'un programme

l'exécution du programme devient indéterministe (non reproductible)

il devient difficile (voire impossible) de raisonner sur l'effet du programme

Danger des interruptions (2)

❖ Exécution pseudo-parallèle

les changements de programme sont déclenchés par des interruptions d'horloge



exemple : deux opérations de dépôt sur un même compte bancaire

activité 1

activité 2

1.1: $\text{courant}_1 = \text{lire_compte} (867A)$

1.2: $\text{nouveau}_1 = \text{courant}_1 + 1000$

1.3: $\text{ecrire_compte} (867A, \text{nouveau}_1)$

2.1: $\text{courant}_2 = \text{lire_compte} (867A)$

2.2: $\text{nouveau}_2 = \text{courant}_2 + 3000$

2.3: $\text{ecrire_compte} (867A, \text{nouveau}_2)$

un scénario d'exécution : 1.1 ; 1.2 ; 2.1 ; 2.2 ; 2.3 ; 1.3

résultat net ?

le compte est crédité de 1 000 € au lieu de 4 000 € !

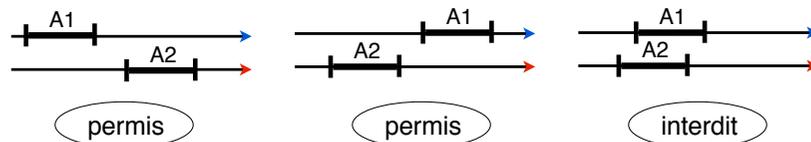
Un problème de synchronisation

❖ Comment éviter la perte de mise à jour ?

un remède : imposer une exécution séquentielle des deux opérations de dépôt (A1 et A2)

soit A1 puis A2, soit A2 puis A1 : tout entrelacement est interdit

autrement dit : une seule des activités (au plus) peut exécuter à un instant donné l'opération en cause (dite section critique)

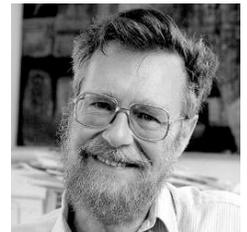
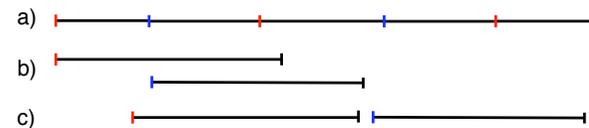


Cette condition s'appelle l'exclusion mutuelle ; c'est le premier problème de synchronisation explicitement formulé (Dijkstra, 1965)

La notion de processus

❖ Un pas vers l'abstraction...

qu'y a-t-il de commun entre les exécutions d'activités suivantes ?



Edsger W. Dijkstra
(1930 - 2002)
crédit : University of Texas at Austin

réponse : les deux activités considérées comme séquences d'actions, indépendamment des conditions de leur exécution.

on arrive ainsi à la notion de **processus** (séquentiel), indépendant du processeur (et des autres ressources)

avantage : séparer les aspects logiques des aspects d'allocation de ressources

exemple : l'exclusion mutuelle est une contrainte logique

Exclusion mutuelle

❖ Spécifications

- un ensemble de processus, une section critique dans chacun ; un processus reste un temps fini dans sa section critique
- ❖ Exclusion : à un instant donné, un processus au plus exécute sa section critique
- ❖ Pas d'interblocage : si plusieurs processus essaient d'entrer en section critique, l'un d'entre eux finira par y parvenir
- ❖ Pas de privation : si un processus tente d'entrer en section critique, il finira par y parvenir

❖ Principes de réalisation

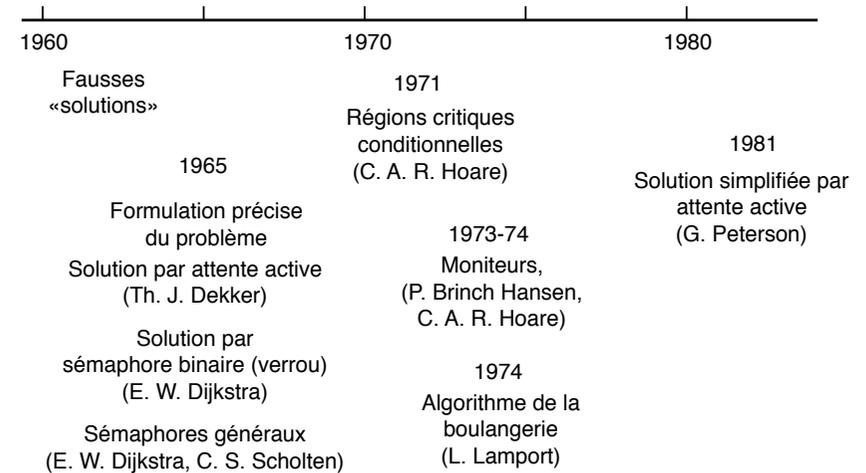
il faut faire attendre des processus. Deux moyens :

- ❖ Attente active : le processus boucle sur un test
- ❖ Blocage : le processus est placé dans un état inactif, où il attend d'être réveillé

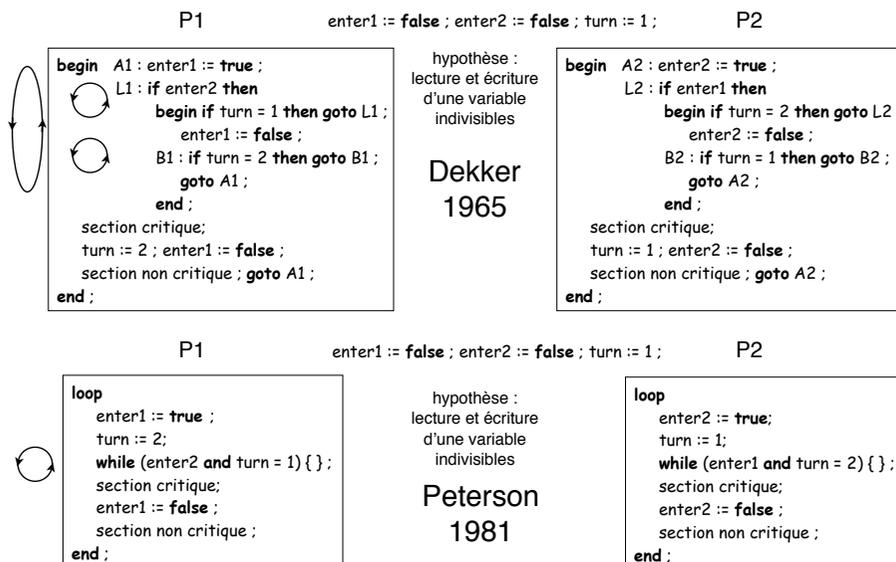
Les débuts de l'exclusion mutuelle

(univers centralisé)

Support matériel
(*test-and-set*)

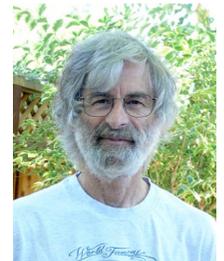


Exclusion mutuelle par attente active (2 processus)



Algorithme de la boulangerie

(Leslie Lamport, 1974)



Principe : chaque processus prend un ticket numéroté (ordre non décroissant)
L'entrée se fait par ordre croissant des tickets (égalité résolue par n° de processus)

```

number[1] := 0 ; ... ; number[n] := 0 ;
choosing[1] := false , ... , choosing[n] := false ;
        
```

```

choosing[i] := true ;
number[i] := 1 + max(number[1], ..., number[n]) ;
choosing[i] := false ;
        
```

```

forall j ≠ i do
  while choosing[j] {}
  while number[j] ≠ 0 and (number[j],j) < (number[i],i) {}
end
section critique
number[i] := 0
        
```

prise du ticket par processus n° i

entrée processus n° i

Intérêt : pas d'hypothèse sur accès atomique aux emplacements de mémoire

Support matériel pour l'exclusion mutuelle

❖ Idée de base : (test + modification) indivisibles

Exemple : `test-and-set (mem, val)`

indivisible `temp := *mem` *mem* : adresse de mémoire
`*mem := val` *val* : valeur
`result := temp` **mem* : contenu de *mem*

Réalisation d'un verrou

`*m := 1` condition initiale

attente active `while test-and-set(m, 0) = 0 {` si libre, prendre le verrou

section critique

`*m := 1` relâcher le verrou

autres opérations : `compare-and-swap`, `fetch-and-add`

Verrous logiciels : sémaphore binaires

Dijkstra, 1965

❖ Idée : supprimer l'attente active

un processus peut transiter entre l'état actif et l'état «bloqué» (inactif)
 sémaphore binaire *s* : une valeur *val(s)* : 0 [occupé] ou 1 [libre] ; deux opérations : *prendre*, *relâcher* ; une queue de processus bloqués sur *s* : *file(s)*

`demander(s)` appelés par processus *p* `libérer(s)`

indivisible `if val(s) = 0 then`
 état(p) := bloqué ;
 entrer p dans file(s)
`else val(s) := 0 ;`

indivisible `if file(s) non vide then`
 sortir un processus q de file(s) ;
 état(q) := actif
`val(s) := 1 ;`

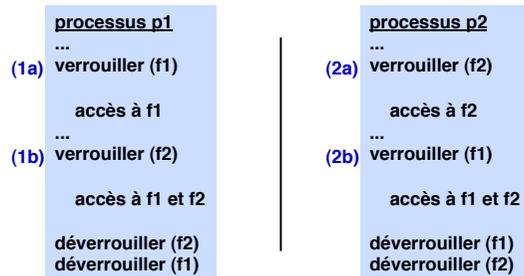
❖ Exclusion mutuelle condition initiale : *val(s) = 1*

`demander(s)` `demander(s)`
 section critique section critique
`libérer(s)` `libérer(s)`

Risque du verrouillage : l'interblocage (1)

Verrouillage de fichiers : *verrouiller(f)* assure l'accès exclusif au fichier *f*, jusqu'au déverrouillage.

Les processus *p1* et *p2* partagent deux fichiers *f1* et *f2*, et les utilisent en accès exclusif.



p1 et *p2* s'exécutent en parallèle ; pour une exécution particulière, la séquence est :

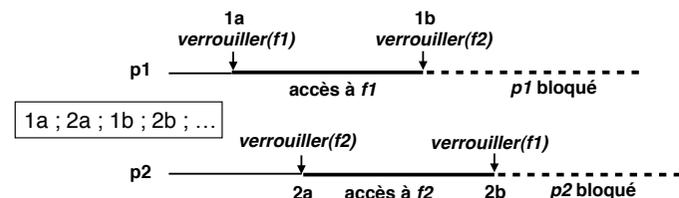
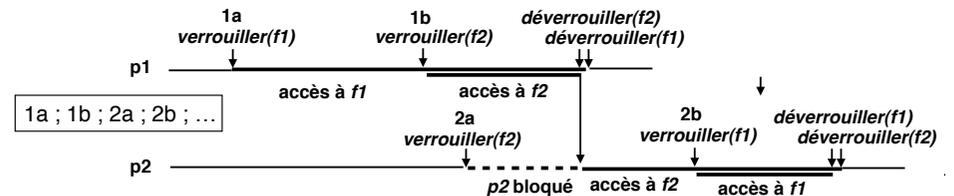
1a ; 1b ; 2a ; 2b ; ...

pour une autre exécution :

1a ; 2a ; 1b ; 2b ; ...

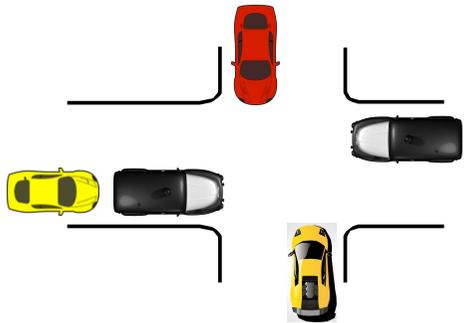
Que se passe-t-il dans chaque cas ?

Risque du verrouillage : l'interblocage (2)



p1 et *p2* sont bloqués indéfiniment car chacun ne peut être réveillé que par l'autre : c'est une situation d'**interblocage**

L'interblocage : prévenir ou guérir



Questions :

- ❖ quelles sont ici les ressources ?
- ❖ comment sortir de l'interblocage ?

Prévention

- ❖ mettre des feux (chaque processus demande en bloc toutes les ressources dont il a besoin) - réduit le parallélisme
- ❖ mettre un rond-point giratoire (tous les processus demandent leurs ressources dans le même ordre, avec priorité)

Guérison

- ❖ on ne peut pas sortir de l'interblocage sans perdre quelque chose

De l'exclusion mutuelle à la synchronisation

- ❖ Les sémaphores compteurs (Dijkstra, Scholten, 1965) généralisent les sémaphores binaires : $val(s)$ est un entier relatif (positif ou négatif)

demander(s)

appelés par
processus p

libérer(s)

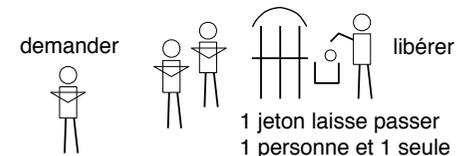
indivisible

$val(s) := val(s) - 1 ;$
if $val(s) < 0$ **then**
 état(p) := bloqué ;
 entrer p dans file(s) ;

indivisible

$val(s) := val(s) + 1 ;$
if $val(s) \leq 0$ **then**
 sortir un processus q de file (s) ;
 état(q) := actif

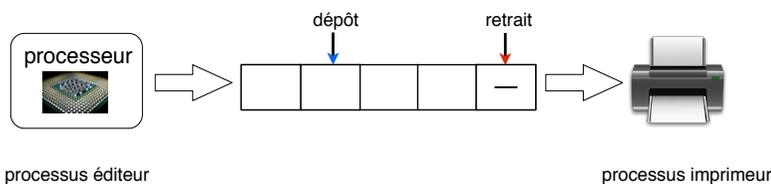
une analogie : le portillon automatique



si $val(s) < 0$: - longueur de la file
 si $val(s) > 0$: nombre de jetons
 d'avance

Un exemple de synchronisation

- ❖ Mémoire tampon pour une imprimante



sémaphore $nvide := 5, nplein := 0 ;$

page := suivante ;
demander ($nvide$) : ←
 entrer (page) ;
libérer ($nplein$) : → **demander** ($nplein$) ;
 sortir (page) ;
libérer ($nvide$) ;
 imprimer (page)

Schéma producteur - consommateur

Noter la symétrie : l'éditeur produit des cases pleines, consomme des cases vides
 l'imprimeur consomme des cases pleines, produit des cases vides

Synchronisation et communication (1)

- ❖ Les messages peuvent réaliser la synchronisation

envoyer(message, destinataire)

recevoir(message) bloquant en attente de message

- ❖ Comparaison avec sémaphores

capacité d'expression équivalente

les messages fonctionnent aussi en univers réparti

- ❖ De nombreuses variantes...

désignation du destinataire

directe (processus), indirecte (boîte aux lettres)

mode de synchronisation

envoi non bloquant (message) ou bloquant (rendez-vous)

persistance des messages

message transitoire (perdu si non attendu) ou persistant (conservé)

Synchronisation et communication (2)

❖ Messages et événements

un événement peut être vu comme un message au contenu vide (la seule information est qu'il s'est produit)
la synchronisation par événement-réaction est apparue très tôt (fin années 50)
à l'époque, outil peu maîtrisé (cf interruptions)

❖ Flots de données

une manière duale de voir la synchronisation (1961)
l'événement : «toutes les données attendues sont arrivées»
déclenche une action (le traitement de ces données)

Un outil de plus haut niveau : les moniteurs

❖ Idée

encapsuler une structure de données
et ses fonctions d'accès
synchronisées
un niveau d'abstraction supérieur
à celui des sémaphores



Tony Hoare
(1934 -)

by Rama, Wikimedia Commons

❖ Exemple : le tampon

2 opérations :

tampon.déposer
tampon.retirer

- ❖ réalisent la synchronisation
- ❖ cachent les détails de la mise en œuvre



Per Brinch Hansen
(1938 - 2007)

Wikimedia Commons

Les moniteurs : structure interne

tampon : moniteur ;

var n : 0..N ;

non_plein, non_vide : condition ;

procédure déposer (m : message) ;

if n = N **then**

non_plein.attendre

n := n + 1 ;

mettre (m) ;

non_vide.signaler ;

procédure retirer (**var** m : message) ;

if n = 0 **then**

non_vide.attendre

prendre (m) ;

n := n - 1 ;

non_plein.signaler ;

initialisation n := 0 ;

À noter :

les conditions
les opérations sur les conditions

attendre : bloquer le processus, qui
entre dans une file d'attente
associée à la condition

signaler : réveiller un processus de la
file associée à la condition

Les procédures du moniteur sont
exécutées en exclusion mutuelle

Bilan des débuts de la programmation concurrente

1965-75

❖ Avancées ...

La notion de processus

Le problème de l'exclusion mutuelle et ses solutions

Compréhension et solution de l'interblocage

Des outils de synchronisation : sémaphores, messages, moniteurs

❖ Limites ...

Preuve de programmes parallèles

première réponse en 1976 (Owicki-Gries)

Intégration dans les langages de programmation

premières réponses à partir de 1975 (Concurrent Pascal, Mesa, ...)

Prise en compte des défaillances

transactions, à partir de 1976

Prise en compte de la répartition

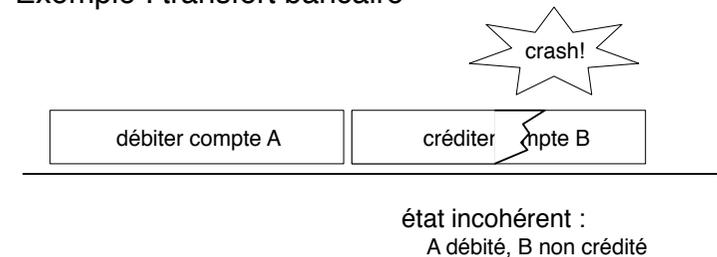
horloges logiques, 1978



Synchronisation en univers réparti et non fiable

Les défaillances, source d'incohérence

❖ Exemple : transfert bancaire



❖ Un remède

rendre indivisible la séquence *débité-créditer*

exécution «tout ou rien»

analogie avec exclusion mutuelle, mais une différence :

en cas de défaillance, il faut restaurer l'état initial
cela revient à «remonter le temps»

Redéfinir l'atomicité

❖ ... vis-à-vis des défaillances

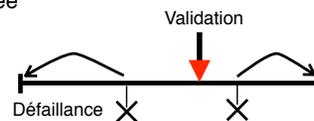
Exécution «tout ou rien»

pas de défaillance : l'action est exécutée

défaillance : on revient à l'état initial

On peut mieux faire ...

point de validation



❖ ... vis-à-vis de la concurrence

Une notion déjà vue

exécution concurrente de A et B :

équivalent à A ; B ou B ; A



Une reformulation

exécution concurrente de A et B :

A n'a pas accès à l'état de B, B n'a pas accès à l'état de A
c'est la notion d'isolation

Les transactions

❖ Dans le contexte des bases de données

Deux propriétés requises pour
une action composée

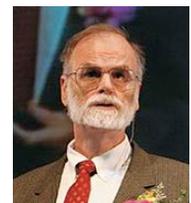
❖ Cohérence

Propriété de l'état, dépendante de l'application, à maintenir

❖ Durabilité (ou permanence)

Après validation, les modifications de l'état sont préservées

Propriété garantie par le support de stockage (mémoire stable)



Jim Gray
1944-2007
©Microsoft

❖ La notion de transaction ACID

Une action composée (séquence d'actions élémentaires) avec

❖ Atomicité (tout ou rien en cas de défaillances)

❖ Cohérence (maintien de propriétés spécifiées)

❖ Isolation (non-interférence de transactions indépendantes)

❖ Durabilité (préservation des résultats après validation)

Réaliser les transactions

❖ Un système de gestion de transactions

Si l'utilisateur assure la cohérence (C), le système garantit atomicité (A), isolation (I), durabilité (D)

La durabilité (D) est assurée par la mémoire stable. Restent A et I

❖ Garantir l'isolation

Isolation = sérialisabilité (équivalence à une exécution séquentielle)

Principe : verrouillage des données (pour la durée nécessaire)

Une propriété : le verrouillage à deux phases assure la sérialisabilité

phase 1 : verrouiller (progressivement) ; phase 2 : déverrouiller

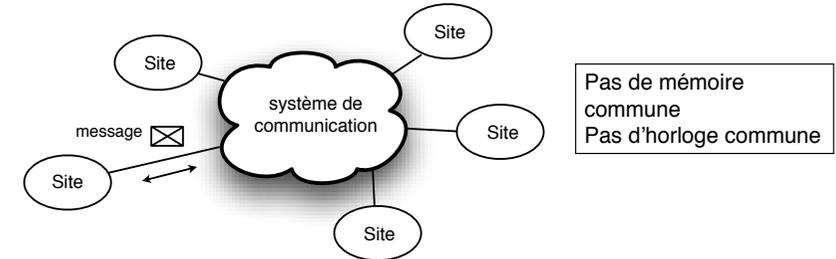
❖ Garantir l'atomicité

Principe : conserver un journal (liste des opérations), pour pouvoir «défaire» en cas de besoin (défaillance)

Le journal est en mémoire stable ; les actions sont enregistrées avant leur exécution

Le temps en univers réparti

❖ Qu'est-ce qu'un système réparti ?



❖ Propriétés du système de communication

Types possibles de défaillances

perte de messages (détectée ou non)

altération de messages, création de messages

Synchrone ou asynchrone ?

durée de transmission bornée ou non

Raisonner en univers réparti

❖ Motivations

On veut coordonner des activités, donc parler d'ordre

On veut définir des propriétés, donc parler d'état

❖ Hypothèses sur la communication

fiable (pas de perte ou d'altération de messages)

asynchrone (pas de borne sur le délai de transmission)

dans la pratique, on fixe souvent une limite

pas nécessairement FIFO

❖ Que veut-on garantir ?

Sûreté (un événement indésirable n'arrivera jamais)

interblocage, incohérence des données, violation de protection

Vivacité (un événement désirable finira par arriver)

un message sera délivré, une ressource sera disponible, un calcul se terminera

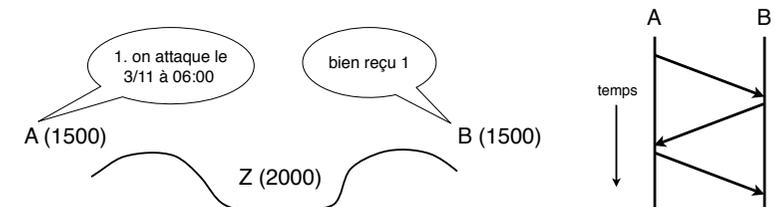
Les limites du raisonnement

❖ De nombreux résultats d'impossibilité

liés au système de communication et aux types de défaillances

❖ Premier exemple : «le paradoxe des généraux» (1975)

si l'émetteur ne peut s'assurer de la délivrance d'un message, l'accord entre deux partenaires est impossible



en pratique : hypothèse de synchronisme (délai de garde)

protocole de l'accord confirmé (chacun voit un aller-retour)

Définir un ordre en univers réparti

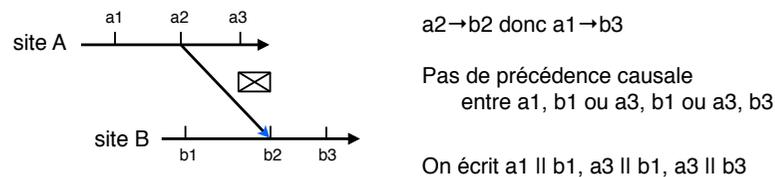
❖ Motivations

- Synchroniser des activités
- Maintenir la cohérence de données

définir un ordre
entre événements

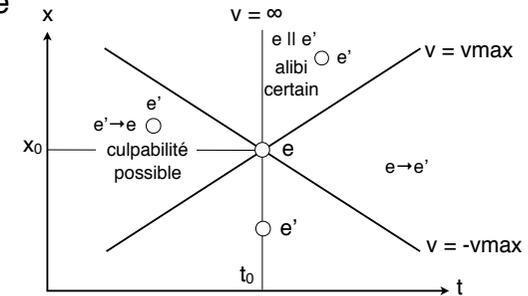
❖ Principe

- Comment définir une relation de précédence pour des événements sur des sites différents ?
- Utiliser le principe de causalité : la cause précède l'effet
- Entre deux sites : l'envoi d'un message précède sa réception



Sur la dépendance causale

❖ Un exemple



- La précédence causale n'indique qu'une dépendance potentielle
- Si $e_1 \rightarrow e_2$, alors il est impossible que e_2 soit cause de e_1 , mais il est seulement possible (non certain) que e_1 est cause de e_2

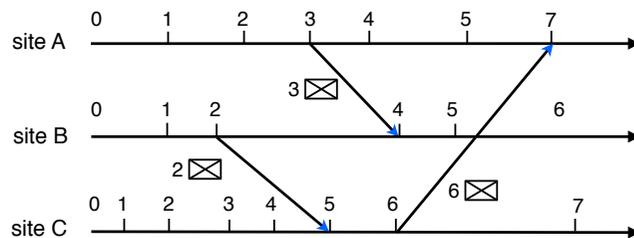
❖ Application

- Recherche de la cause d'une défaillance

Datation en univers réparti

❖ Les horloges logiques (Lamport, 1978)

- Un compteur par site, incrémenté à chaque événement
- Les messages sont estampillés par l'heure d'envoi
- À la réception d'un message, le compteur local est mis à jour pour respecter la causalité



- Sur chaque site, l'heure d'un événement est la valeur de son compteur (horloge logique, ou HL)

Les horloges logiques

❖ Propriétés

- Les horloges logiques traduisent (faiblement) la dépendance causale

Si $e_1 \rightarrow e_2$, alors $HL(e_1) < HL(e_2)$

Mais l'inverse n'est pas vrai

Si $HL(e_1) < HL(e_2)$ alors e_2 ne précède pas e_1 :

ou bien $e_1 \rightarrow e_2$, ou bien $e_1 || e_2$

- Pour caractériser fortement la dépendance, il faut des horloges plus raffinées (horloges vectorielles ou matricielles)

❖ Usage

- File d'attente répartie
 - Algorithmes de synchronisation (exclusion mutuelle, diffusion)
 - Mise à jour cohérente de données dupliquées
- Gestion de caches
- Génération de noms uniques

Cohérence de données réparties

❖ Définir la cohérence ?

un problème en soi...
... de nombreuses définitions

❖ Quelques exemples

cohérence stricte

une lecture rend le résultat
de la «dernière» écriture

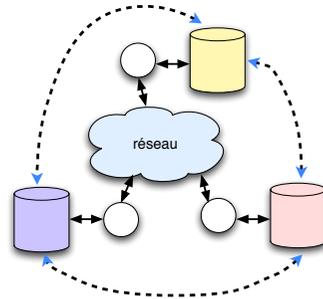
cohérence à terme

en l'absence d'écriture, tous les
exemplaires «finiront par» devenir identiques

entre temps, une lecture peut renvoyer une valeur «ancienne»

cohérence causale

si l'événement e1 (écriture ou lecture) précède causalement
l'événement e2, alors e1 est vu avant e2 sur tous les sites



De la synchronisation à la décision concertée

❖ Motivations

De nombreuses situations exigent qu'un ensemble de processus
prennent une décision concertée

validation de transactions réparties

diffusion d'informations dans le même ordre pour tous les récepteurs

détermination (unique pour tous) de la composition d'un groupe

Le consensus est un des problèmes clé des systèmes répartis

très simple en l'absence de pannes

très difficile avec pannes et communication asynchrone

❖ Spécifications du consensus (informelles !)

Chaque processus propose une décision (ex : choisir une valeur)

Chaque processus décide

Tous (sauf ceux défaillants) prennent la même décision (l'une de celles
proposées)

La décision est prise en temps fini

Réaliser le consensus

❖ Un résultat d'impossibilité (Fischer, Lynch, Paterson, 1983)

Dans un système asynchrone avec pannes franches, le
consensus est impossible par un protocole déterministe,
dès qu'un processus est en panne

Explication sommaire : en asynchrone, impossible de
distinguer un processus lent d'un processus défaillant

❖ Solutions pratiques

Détecteurs de pannes (Chandra, Toueg, 1991)

Le consensus est possible avec un détecteur de pannes même
imparfait ; on essaie d'approcher un tel détecteur

Paxos (Lamport, 1989 ; préfiguration par Oki et Liskov, 1988)

Utilise un coordinateur, remplacé si défaillant

Dans tous les cas, hypothèse de synchronisme (délai de
garde), qui peut échouer

Paxos (vue très sommaire)

❖ Dans un monde parfait...

On choisit un processus coordinateur

Celui-ci recueille les votes des participants et propose une décision

Si tout va bien, la décision proposée est adoptée à la majorité

❖ ... mais les défaillances arrivent

Cas d'échec possibles (pas de décision prise)

Panne du coordinateur

Impossibilité d'obtenir une majorité

panne des processus participants

communication asynchrone

En cas d'échec

On choisit un autre coordinateur

Si une décision a été proposée antérieurement, elle est préservée

Décision si un coordinateur garde une majorité pour le temps de 2
aller-retours

Paxos

❖ Paxos en pratique

Des hypothèses de fonctionnement très faibles

pannes avec réinsertion

perte de message

Une mise en œuvre délicate et complexe

❖ Extensions

Consensus avec pannes «byzantines» (comportement quelconque, y compris malveillant)

Une extension de Paxos (Castro, Liskov, 2001), nombreuses améliorations ultérieures, sujet «chaud»

❖ Exemple d'utilisation

Gestion des données réparties dans Google

Maintien de la cohérence de données dupliquées en cas de pannes

Quelques tendances actuelles de la synchronisation répartie

❖ Nouveaux paradigmes

exemple : Map-Reduce (réduire les occasions de synchronisation)

❖ Retour en grâce des événements

stimulé par les nouvelles applications (graphiques, etc.)

modèles et outils pour la coordination à grande échelle

❖ Évolution du modèle de transactions

le Web comme grande base de données

peu structurée, évoluant en permanence

nouvelles visions des transactions

transactions longues

au delà du schéma ACID

❖ Vers des applications «réelles» du consensus

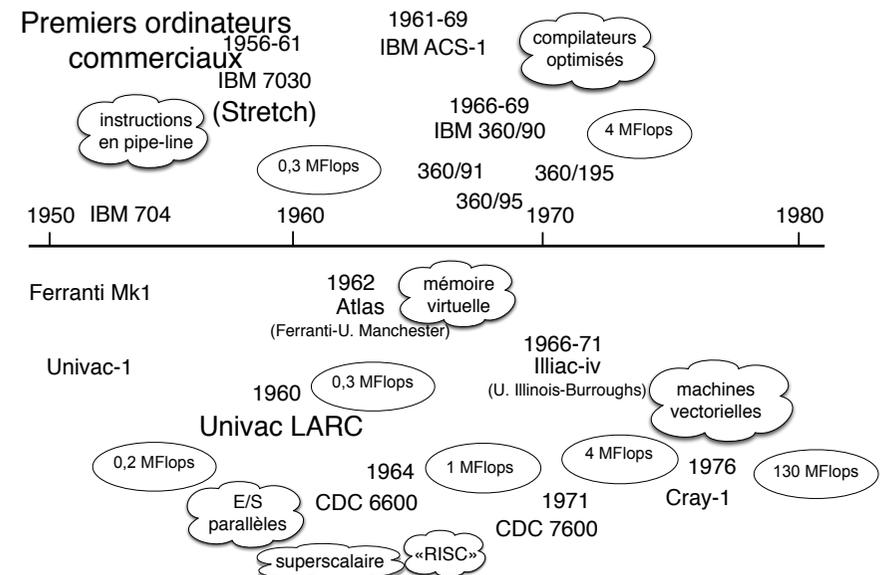
cohérence de données

communication de groupe

X

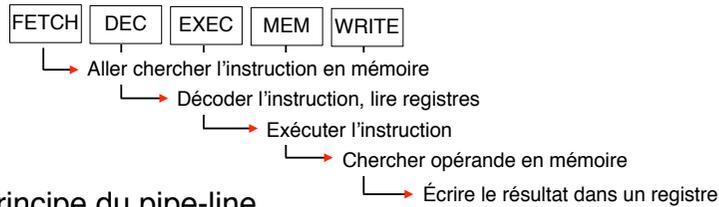
Le parallélisme dans les premiers supercalculateurs

Les premiers «supercalculateurs»

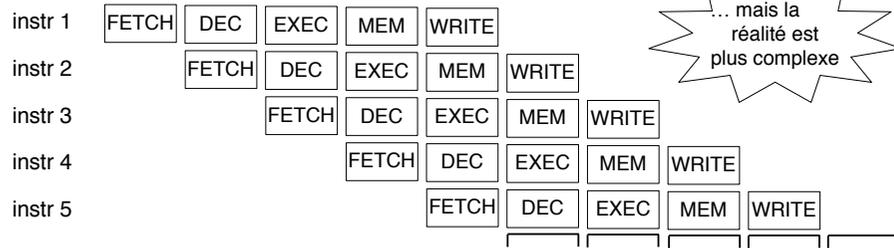


Instructions en pipe-line

❖ Les étapes d'une instruction (simplifié !)



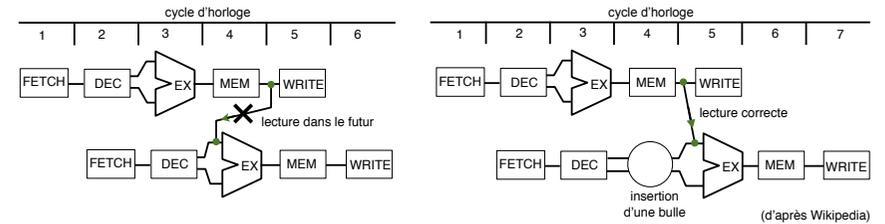
❖ Principe du pipe-line



Difficultés du pipe-line (1)

Le fonctionnement du pipe-line peut être perturbé pour diverses raisons, qui empêchent le déroulement continu des instructions.

❖ Conflits d'accès aux données



ici : conflit *read after write* (RAW), le plus fréquent
autres conflits possibles : WAW, WAR

Solutions :

- insérer des «bulles» (mais retarde l'exécution)
- ordonnancer dynamiquement (changer l'ordre d'exécution)
- se reposer sur les compilateurs

Difficultés du pipe-line (2)

❖ Conflits d'accès aux ressources

source d'attente («bulles»)

❖ Instructions de branchement conditionnel

si on attend l'évaluation de la condition, on retarde l'exécution
une voie d'approche : faire une hypothèse sur le résultat
toujours la même (ex : pas de branchement)
prévision fondée sur l'histoire antérieure

donc :

- si l'hypothèse est juste, on ne perd rien
- si elle est fausse, il faut annuler le travail fait à tort (bulle)

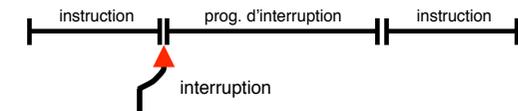
autre voie :

- changer l'ordre des instructions (si c'est possible) pour «avancer» l'évaluation de la condition

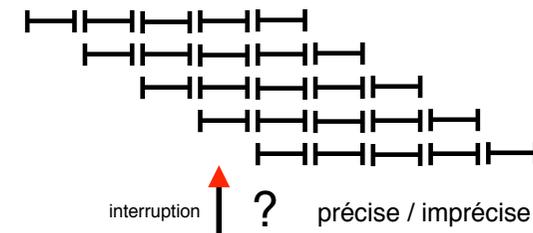
Difficultés du pipe-line (3)

❖ Interruptions

dans une architecture sans pipe-line : «points interruptibles»
(en général entre deux instructions)



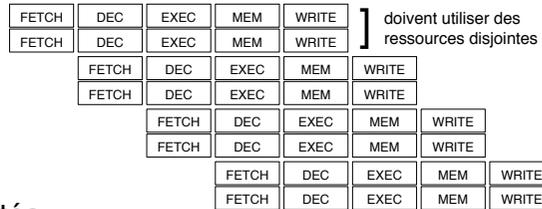
où sont les points interruptibles dans un pipe-line ?



Architectures superscalaires

❖ Une autre source de parallélisme

lancer plusieurs instructions à la fois
 les ressources doivent permettre leur exécution en parallèle
 peut être combinée avec un pipe-line



❖ Difficultés

risques accrus de retards dus aux dépendances
 d'où réorganisation dynamique de l'exécution des instructions

Machines parallèles emblématiques des années 1960-70

IBM 7030 «Stretch» (1956-51)

❖ Un objectif ambitieux

100 fois la vitesse du 704

❖ Des avancées techniques

première machine IBM à transistors
 une des toutes premières pour les
 instructions en pipeline

❖ Une équipe talentueuse

Steve Dunwell, Gene Amdahl (au début)
 «débutants» prometteurs : Fred Brooks, John Cocke

❖ Un échec commercial...

performances limitées (30 x 704 au lieu de 100), 9 exemplaires
 vendus

❖ ... mais une expérience précieuse pour la suite



Console de maintenance
de l'IBM 7030
Musée des Arts et Métiers
©2006 David Monniaux

CDC 6600 (1962-64)

❖ Le premier «supercalculateur»

1 MFlops, > 100 exemplaires

❖ Des avancées décisives

entrées-sorties parallèles
 (10 processeurs périphériques
 multiplexés)

jeu d'instructions «RISC» avant l'heure
 architecture «superscalaire» avant l'heure
 ordonnancement dynamique
 cache d'instructions

❖ Un architecte de génie

Seymour Cray (1925-1996)



CDC 6600
CC BY 2.0 2006, Jitze Couperus



Seymour Cray
Wikipedia Commons