

Fondements algorithmiques des systèmes distribués

Chapitre 1 : Principes et mécanismes de base des systèmes distribués

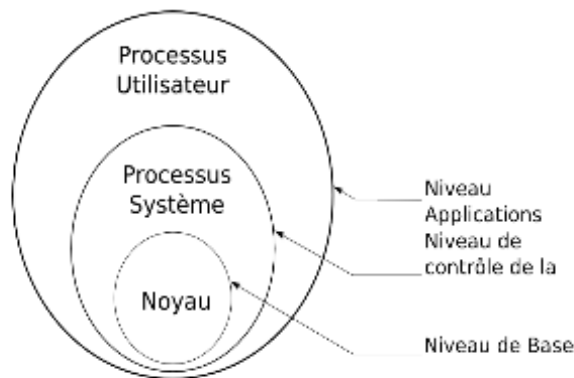
1 Introduction

1.1 Rôles d'un SD

Système réparti = système distribué

Système mono site (d'exploitation) : 2 grandes parties :

- le noyau
- les processus (système et utilisateur)



Les différents niveaux d'un système d'exploitation

Système distribué : machines sur plusieurs sites à faire fonctionner ensemble

Système dans lequel les processus vont communiquer entre eux, sur plusieurs machines différentes, pour réaliser une application.

Système d'exploitation distribué \neq SD : doit faire le travail d'exploitation.

Assuré les mécanismes de base des systèmes d'exploitation sur plusieurs machines (E/S, file system, ordonnancement).

Problème général avec les systèmes distribués : savoir si le processus est terminé (problème de variation de bande passante dans le réseau par exemple) \rightarrow Mécanismes de contrôles liés à la distribution = nouveaux problèmes.

Appli a récupérer un ensemble de processus défini par qqn avec des contraintes, assurer quelle fonctionne correctement.

Contrôle de la
distribution

Base

Rôle d'un OS distribué : offrir aux applications les moyens de s'exécuter.

Assurer le partage et la communication d'informations entre les applications.

Assurer l'exécution parallèle.

Partager les ressources physiques et logiques sur un ensemble d'utilisateurs.

Mécanismes à mettre en place :

L'ordonnancement, placement de processus.

Synchronisation, terminaison.

Désignation, nommage, partage d'informations.

Contrôle de l'état global.

Cohérence.

Diffusion d'informations.

Exclusion mutuelle.

Reprise sur panne.

Sécurité.

Administration.

Transactions.

Communication entre deux processus :

sur la même machine : mémoire partagée à architecture fortement couplée, lorsque les processus communiquent via une mémoire globale.

Durée d'une communication est négligeable.

Sur des machines différentes : par messages à architecture faiblement couplée, lorsque les processus communiquent par échange de messages (via le réseau).

Il faut tenir compte de la durée des communications (en plus possibilité de pertes de messages).

Système symétrique (Turing) \neq système distribué.

Architecture parallèle avec mémoire globale à le système va être symétrique.

1.2 Caractéristiques des SD

On étudiera que des systèmes distribués faiblement couplés.

- Absence d'état global

Comment faire pour savoir en quel état doit reprendre le processus après une panne. On n'est pas capable de savoir ce qu'il y a sur le réseau.

Un processus ne connaît que les événements qu'il a provoqués sur son site.

Un processus ne connaît que les messages qu'il a envoyés ou reçus.

Il n'y a pas d'ordre temporel naturel entre les événements.

Il est difficile de synchroniser deux horloges (impossible)

Il n'y a pas d'échelle.

- Distribution des données

Deux cas :

- distribution inhérente aux problèmes : le concepteur se voit imposé, par exemple, une distribution géographique (disques de Bases de données sur plusieurs sites, ...)
- distribution inhérente à la mise en œuvre : les utilisateurs veulent pouvoir travailler sur des sites distants.

D'où

- duplication : une donnée peut se trouver dupliquée sur plusieurs sites => problème de cohérence
- partitionnement : une grande BD peut être coupée en plusieurs sous-BD sur des sites différents => problème de désignation (comment retrouver une donnée de la base : il faut d'abord savoir sur quel site elle se trouve)

Bien évidemment, les deux peuvent se faire en même temps.

- Contrôle

On dira qu'il y a contrôle distribué lorsqu'il n'y a pas de relation hiérarchique statique entre les processus : il n'y a pas de processus qui joue de rôle particulier a priori. Ainsi, il n'y a pas de maître qui assure en permanence le contrôle global.

Pourquoi :

- tolérance aux fautes faible voire nulle : si ce processus s'arrête => n du système ;
- goulet d'étranglement.

Il se peut que pour certaines fonctions du système, il y ait nécessité d'un maître : on utilisera alors un mécanisme d'élection.

2 Éléments de base des SD

2.1 Protocole

Un protocole déni le comportement d'un processus vis-à-vis d'autres processus : on va s'intéresser principalement (et presque uniquement) aux réactions du processus à la réalisation d'événements : réception d'un message, réalisation d'une condition, émission d'un message, etc. :

P_i : Processus réserve et utilise une imprimante

Demande : DEM(1)

- émettre la demande à tous les autres processus
- compteur nb de processus $\text{compt} = N - 1$
- attente

* Sur réception d'une demande DEM(j)

* Suivant l'état de P_i

- LIBRE : répondre (OK) à P_i
- OCCUPE : NOK
- ATTENTE : si $(i < j) \Rightarrow$ NOK

* Sur NOK(j)

/* recommence le protocole demande () */

* Sur OK(j)

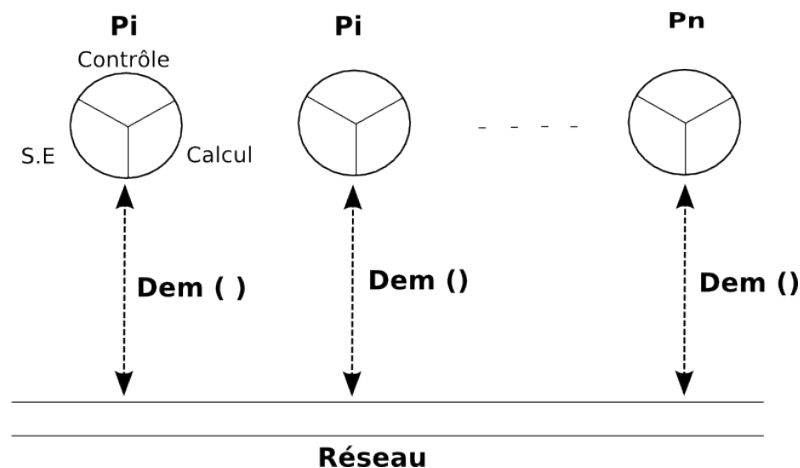
Compteur de processus $\text{compt} --$

si $(\text{compt} == 0) \Rightarrow P_i$ passe à l'état OCCUPE

- On ne s'intéresse que très peu à ce qui se passe en local. Et on suppose que localement, on dispose de ressources logicielles pour réaliser les opérations. La seule contrainte est que le processus dispose de toutes les données nécessaires aux calculs.

- On considère que l'envoi et la réception de message est réalisé par le système d'exploitation

2.2 Processus



2.3 Liaisons

- structure de liaison intersites.
- comportement de liaisons.

2.3.1 Structure des réseaux "virtuels" (propriétés structurelles)

- maillage complet
- hiérarchie
- étoile
- anneaux (unidirectionnel, graphe de communication des sites)

2.3.2 Comportement du réseau (propriétés comportementales)

H1 - Les messages ne sont pas dupliqués.

H2 - Un message, s'il arrive, n'est pas altéré.

H3 - Pas de déséquencement.

H4 - Le délai d'acheminement d'un message est fini. Tout message arrive en un temps fini.

H5 - Le délai d'acheminement d'un message est borné.

H1+H2+H3+H4 = FIFO

Il faudra toujours préciser les propriétés comportementales du réseau de communication interprocessus.

2.3.3 Trafic

Ordre d'optimisation :

- Nombre d'échange de messages.
- Taille des messages.
- Calculs locaux.

3 Techniques d'implantation des protocoles

Pi0 veut diffuser (i=1)

Pi0 envoie M(i=1) à suivant (Pi0)

Pi :

- Lorsque Pi reçoit M(m=k,j)

* si i==j => FIN

* si i!=j

// toutes M

envoie M(n=k,j) à suivant Pi

3.1 Jeton circulant

3.2 Calcul diffusant

4 Conclusion

La construction d'algorithmes ???

Chapitre 3: Partage et gestion de données distribuées

3.1 Introduction

- nommage / désignation => comment accéder physiquement à la donnée
- cohérence des données => comment ...

3.2 Désignation

3.2.1 Nommage des données

Nom interne d'un donnée :

information minimale nécessaire à un Système d'Exploitation pour retrouver la donnée

- ex:
- fichier => inode
 - machine => adresse Ethernet
 - service distant => adresse IP + port

Désignation de nommage :

- > associe des aux objets
- <- interpréter un pour retrouver l'objet

Nom symbolique <-> Nom interne

- nom symbolique $_ >$ 1 seul objet
- permet de retrouver l'objet ou qu'il soit

ex: on concatène le nom symbolique local avec le nom symbolique de la machine pour obtenir le nom symbolique global.

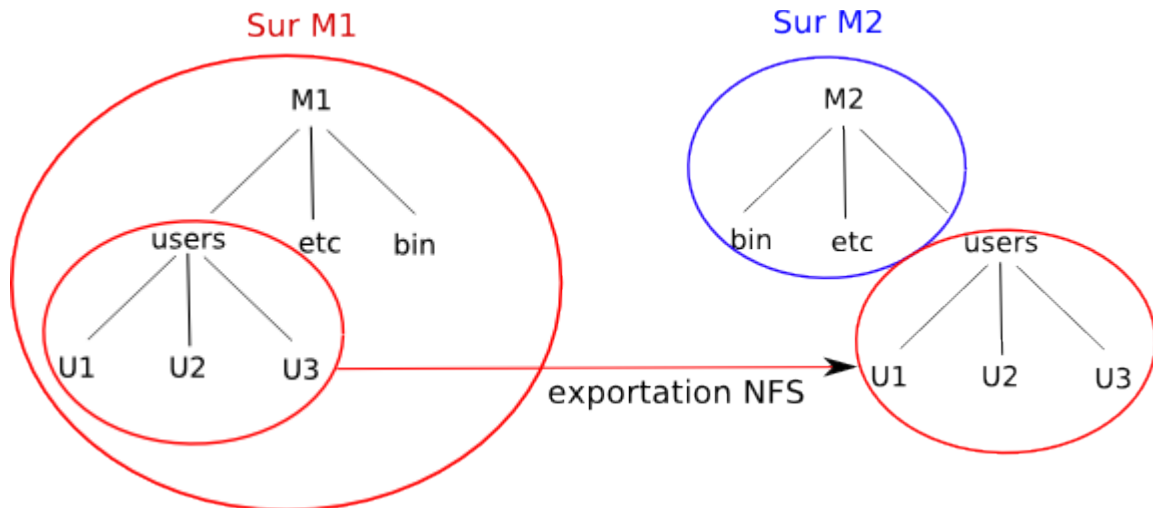
=> Problème lors de la migration des objets

3.2.2 Migration et noms internes

Migration déplacement d'un objet d'une machine vers une autre.

- > Lieu de poursuite -> objet "peu mobile"
- > Objet mobile -> base de données ex: serveur de nom
- > Diffusion

Exemple d'exportation nfs :



Montage des partitions :

Sur M1 :

```
$ mount /dev/hda1/ /users
```

```
$ mount /dev/hda2/ /
```

Sur M2 :

```
$ mount -t nfs //M1:/users /users
```

Selon la version de Linux on utilisera plutôt share (solaris) ou fstab

share => exportfs

fstab => export

Accès :

```
$ cat /home/master 1/f1
```

M2 doit faire un appel système d'interprétation pour connaître le contenu de /users,

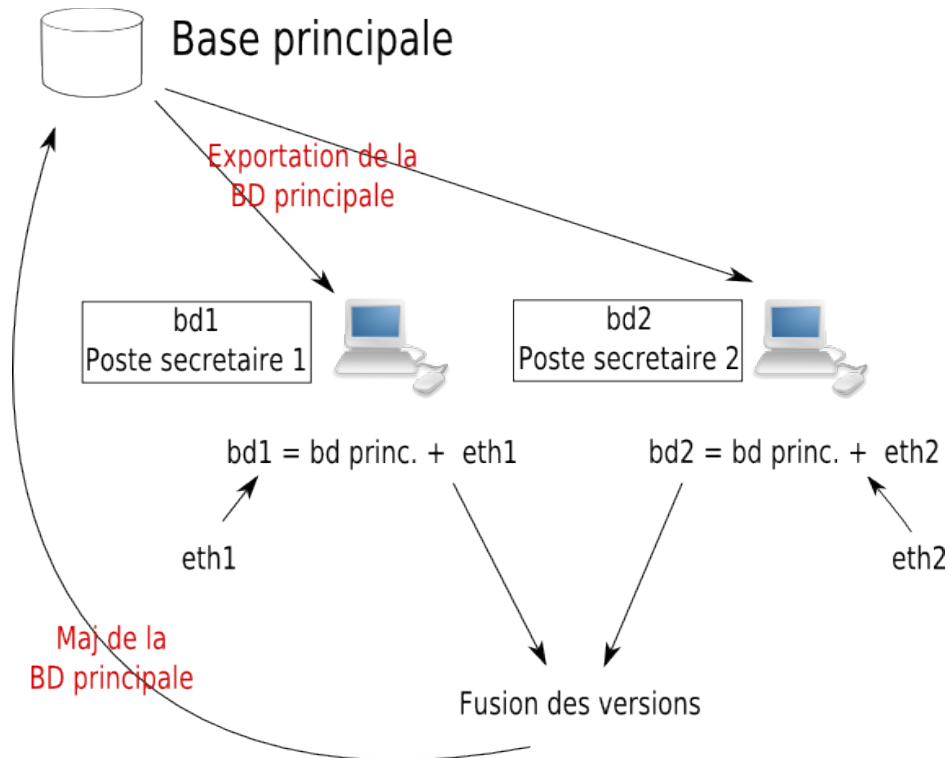
Le système va essayer de contacter M1: pour cela il va envoyer un message à M1 en utilisant un service (demon) mis en place auparavant => M2 envoie read sur le nfsd et M1 lui renvoie les informations

nfsd ne fait rien tout seul il attend des messages tels que read, write.

3.3 Cohérence des données

3.3.1 Définition

Utilisation du cache, cohérence faible.



On ne veut pas sauvegarder à chaque fois les données dans la base principale, on a donc bd1 et bd2 qui sont des copies de la base principale mais qui ne contiennent pas forcément les dernières données à jour, d'où une cohérence faible. Pour compenser il faut faire une mise à jour chaque soir pour réorganiser les données et que chaque base reçoive un contenu identique.

Pourquoi utiliser la cohérence faible ?

Car ça simplifie les manipulations sur les bases séparées, et utilisées par des utilisateurs différents.

Cohérence faible :

- une copie peut être temporairement incohérente
- l'ordre des mises à jour d'une information n'est pas nécessairement celui de sa modification première.

Avantage => mécanisme simple.

Inconvénient => une copie peut être incohérente à un moment donné.

Cohérence forte :

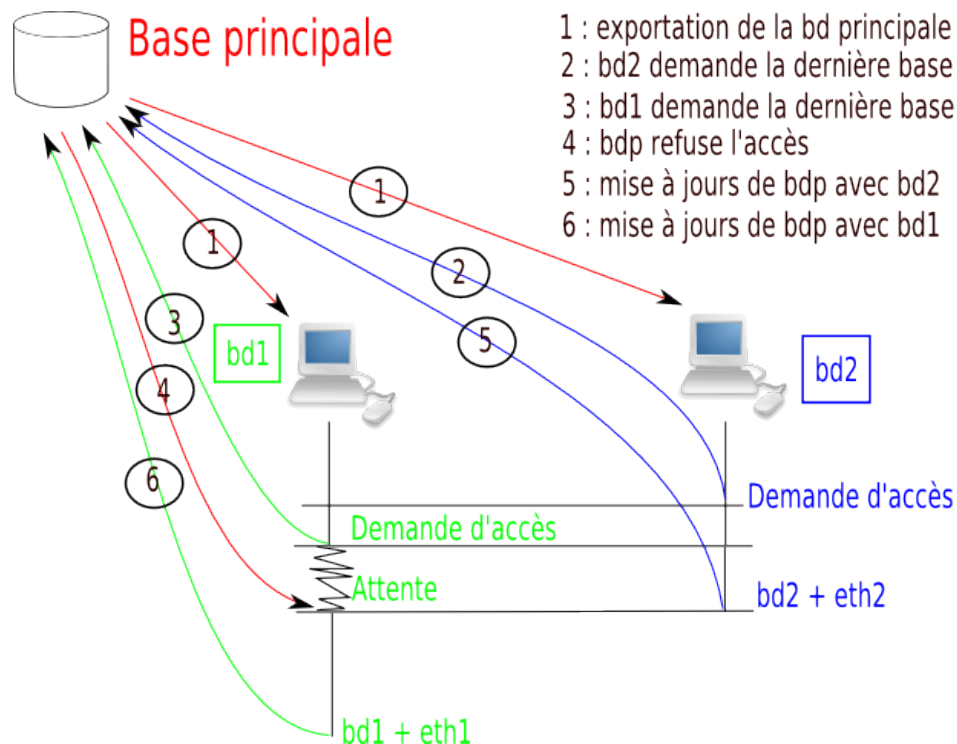
Deux méthodes :

- Centralisé :

- 1) on vérifie que la base de donnée locale est la bonne => on vérifie que notre n° de version correspond à celui de la base principale.
- 2) on doit mettre à jour sur la base principale la donnée modifiée, pour éviter les accès simultanés et des modifications de la donnée en même temps, on verrouille la donnée quand on

effectue la vérification lors de l'étape 1).

Inconvénient : temps d'attente du au verrouillage de la donnée (exclusion mutuelle), et système centralisé.



- Distribuée :

On numérote les modifications de la donnée

Quand on veut accéder à la donnée on fait une demande au serveur et on attend l'acquittement

Si il y a plusieurs versions, on envoi celle dont le N° de modification est le plus grand (code opération). La donnée est renvoyée à la base locale en même temps que l'acquittement.

=> **Cohérence forte** : Chaque fois qu'un site accède sa copie, la valeur de cette copie doit refléter le résultat de toutes les modifications antérieures.

Avantage : Pour chaque accès à une copie de la donnée, sa valeur correspond bien aux résultats des dernières modifications effectuées.

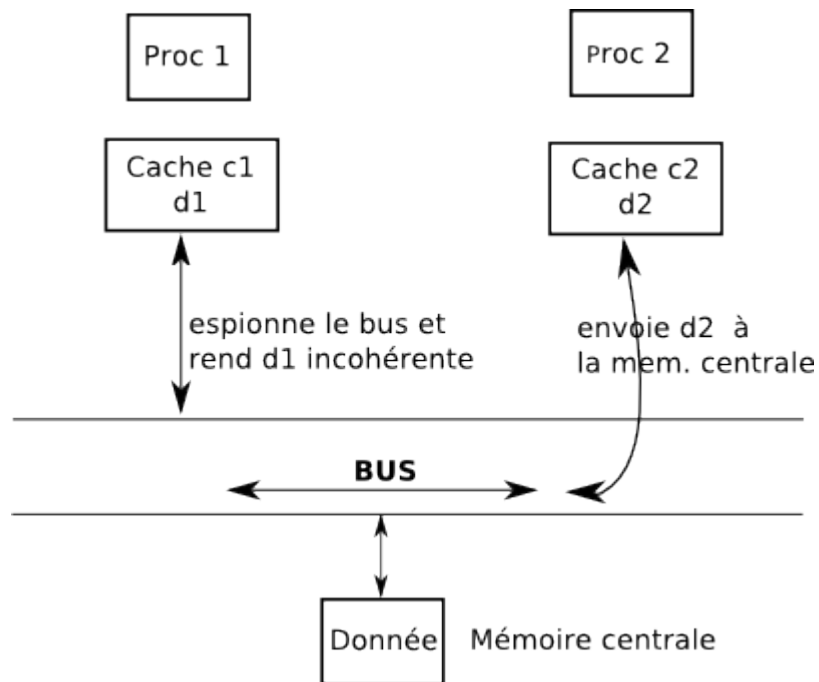
Inconvénient : Echanges de messages (demandes/acquittement) => temps perdu

Evénements:

S'il y a un lien de causalité entre deux événements, je veux que cela influe sur l'ordre des événements que la donnée devra traverser.

Cohérence causale : Si deux événements de modification peuvent être causalement ordonnés alors les prise en compte doivent être fait dans cet ordre.

3.3.2 Cohérence et cache



3.3.3 Détection de l'incohérence

Avant on définissait un protocole qui maintenant la cohérence, cependant il n'est pas forcément tjrs utile que ce protocole tourne tout le temps, on peut en fonction de la probabilité d'une incohérence détecter et corriger l'incohérence.

Mécanisme de détection de l'incohérence :

Précédemment dans le schéma distribué, la tolérance au pannes était nulle, si on perdait un message d'acquittement le système plantait.

On veut permettre à l'application de continuer à travailler malgré une interruption, on a la détection d'une incohérence et on veut donc reconstruire la donnée (la détection n'a d'ailleurs de sens que si on est capable de reconstruire la donnée).

Chapitre 5 : Exclusion mutuelles

5.1 Rappel

5.1.1 Propriété

Définition :

- une section qui correspond à du code auquel l'accès doit se faire de façon exclusive.
- le protocole d'exclusion mutuelle a pour but de garantir cette exclusivité, à un instant, au plus un processus peut se trouver en Section Critique (SC)

Propriété :

* **Atteignabilité :** Si plusieurs processus sont bloqués en attente de la Section Critique alors que aucun processus n'est en SC (la SC libre)

Alors un de ces processus entrera en SC en un temps fini (*le plus court possible indépendamment de l'application*).

* **Progression :** Un processus qui demande à entrer en SC, y parviendra dans un temps fini (*temps qui peut être grand ou petit mais dépend de l'application*).

* **Indépendance :** indépendance entre partie(s) conflictuelle(s) et non conflictuelle(s) : un processus hors de la SC (ou ne demandant pas) ne doit pas influencer sur le protocole d'Exclusion Mutuelle.

* **Banalisation de la solution :** il n'y a pas de processus privilégié.

5.1.2 Monoprocessus

Différentes solutions :

- variable de condition (*variable globale*)
- variable "test & set" (*on test puis modifie une variable (une seule)*)
- moniteur
- sémaphore (*2 opérations : si valeur 1 on peut décrémenter on entre en SC si la valeur 0 on reste bloqué*)

Pour être sûr de ne pas être interrompue le plus simple est de bloquer de le bus, on masque des interruptions, cependant ce système est incompatible avec les solution citées ci dessus en particulier les sémaphores qui ne partagent pas le bus. Ces solutions sont ainsi incompatibles avec des architectures distribuées.

Solutions totalement distribuées => démon

5.2 Algorithme de la boulangerie

5.2.1 Principe

-> un processus S1 voulant entrer en SC s'affecte un numéro $N_i = \max(N_{j \neq i} + 1)$

-> puis Si attend que $N_i = \min(N_{j \neq i} - 1)$
 -> Lorsque Si quitte la SC, $N_i = -1$

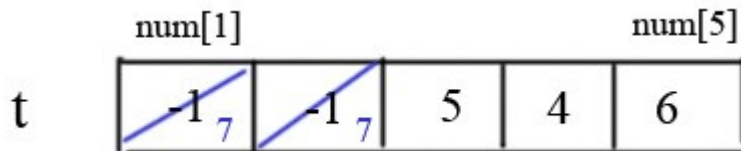
Dans chaque processus, on dispose d'une table $num[i..N]$ initialisé a (-1), on veut avoir dans la table les numéros des processus en attente.

(On a besoin de maintenir cette table à jour uniquement lorsque l'on veut entrer en SC)

-> $dem_num(j)$ (demande le numéro du processus j) \Leftrightarrow if ($num(j) \neq -1$)
 ->

```
Entrer SC() {
  num[i]=max(num[j≠i])+1
  Attendre que (num[i]=min(num[j≠i]) avec num[j]≠-1 )
}
sortie SC() { num[i]=-1 }
```

Exemple:



t=2 S2 veut entrer en SC il s'affecte 7
 S2 test num[3] -> 5
 S2 test num[4] -> 4

t=3 S1 veut entrer en SC il s'affecte 7
 S2 test num[5] -> 6
 S1 test num[2] -> -1 => S1 se met dans num[2] et affecte 7
 S2 test num[1] -> -1 => S2 se met dans num[1] et affecte 7

Problème il y a désormais deux case initialisées à 7

Solution : lors de l'affectation on ajoute le num à la valeur à affecter => S1 on affecte 7+1

- Accès au tableau \leftrightarrow correspond à un ou plusieurs messages
 - > attente active avec un grand nombre de messages
 - > attente passive avec "inscription" (cette solution réduit le nb de message mais nécessite que le client s'identifie)

5.3 Exclusion mutuelle basée sur un jeton

- Cas de l'anneau : trivial
- Cas général :
 - Hypothèse : P_i dispose d'un moyen d'envoyer un message à tous les autres processus
 - * Pas de perte de message.

Principe du protocole :

Idée : Un jeton unique circule entre les processus, un processus ne peut entrer en S.C. (Section critique) que s'il dispose du jeton

Mise en place : * P_i veut entrer en SC, il diffuse une requête vers tous les sites (lui compris)

* P_i reçoit une requête
> il dispose du jeton

Si P_i n'est pas dans la SC, il l'envoi au demandeur

Sinon P_i mémorise la requête
-> il ne dispose pas du jeton => il ne fait rien (NOP)

* P_i reçoit le jeton (*qui lui est destiné*) => P_i entre en SC

* Lorsque P_i quitte la SC()

Si P_i a au moins une requête mémorisée, il envoi le jeton à un des processus en attente

Sinon NOP

Chaque site mémorise la liste des sites ayant fait une demande de jeton, et le nombre de demandes pour chaque site.

Pour éviter de rester bloqué on va mémoriser dans le jeton la liste des sites ayant reçus le jeton ainsi que le nb de fois ou ils l'ont reçus. Dès qu'un site reçoit le jeton il s'ajoute à la liste sur le jeton.

Quand un site à terminé il compare la liste des demandes et la liste des reçus et envoi le jeton à un des sites qui a plus de demande que de réception.

Explications Algorithme:

- Sur réception d'une demande de P_j , P_i exécute $DEMi[j]++$;
- Lorsque P_i quitte la SC, $JETON.SATIF[i]++$;

Si $DEMi[j]==JETON.SATIF[j] \quad \forall j$
> NOP

Sinon
for ($j=(i++)\%N$, $j!=i+1$, $j=(j+1)\%N$)
>_Ca implique que P_i doit avoir le jeton et que P_i n'est pas en SC

Si $JETON.SATIF[j]$
> => il faut que ca soit Atomique
envoi JETON à
break

Sinon NOP

Si P_i a encore le jeton et que $DEMi[i]> JETON.SATIF[i]$
> Alors il se l'envoi

Problème :

- Tolérance aux pannes nulle : si perte de message
- Coût en $(N+1)$

5.4 Exclusions Mutuelles par liste d'attente repartie

Réseau FIFO :

Principe : on va dater les demandes

Chapitre 9 : Ordonnancement et placement de processus

9.1 Introduction

Une Application est un ensemble de tâches.

=> organiser au mieux l'exécution de cet ensemble de façon à ce que l'exécution globale soit le plus rapide possible.

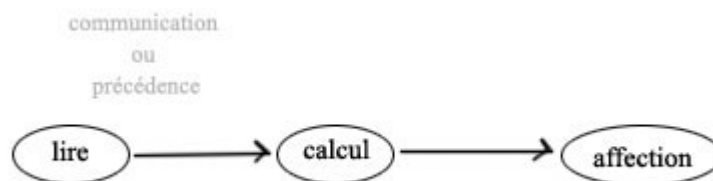
Hypothèses:

Hyp 0 : le nombre de tâches est fixe

Hyp 3 : les communications entre tâches ne peuvent avoir lieu qu'à la fin / ou au début des tâches (réception). Et les Entrée/Sorties sont pas préemptives.

(Préemptive = une fois sur le processeur le processus y reste jusqu'a la fin)

On peut truander un peu pour que des tâches préemptives puissent quand même être exécutées, pour cela on leur donne une priorité telle que dès qu'elles ont fini leur E/S et qu'elles sortent, on les recharge en processeur, ainsi on conserve la non préemptivité.



Hyp 1 : Exécution statique, on connaît a priori :

- l'ensemble des tâches
- leurs durées
- les liens de communications / précédences entre elles

Hyp 2 : Durée des tâches invariante

(quelque soit le processeur, si on lance 10X la même tâche on obtiendra toujours le même temps)

=> 2 techniques : * durée théorique, on calcul.

* on fait du profiling, on conserve les durées lors des sessions précédente et en fonction de celles-ci on fait une moyenne

Hyp 4 : Les communications ne prennent pas de temps

=> 2 techniques : * Soit on n'a pas de communication.

* Soit toute communication ce fait entre deux processus sur le même processeur.

Hyp 5 : On dispose de suffisamment de processeurs

=> on n'a pas a s'occuper du nombre de processeurs, quelque soit le moment si on a besoin de X processeur son aura X processeurs disponibles.

Hyp 6 : Pas de priorité entre les tâches

Hyp 7 : Les ressources sont suffisantes (RAM, etc ...)

A partir de ces Hypothèse on en déduit l'algorithme suivant :

Un ordonnancement est une relation ORD qui à chaque tâche T associe une date de début et un processeur

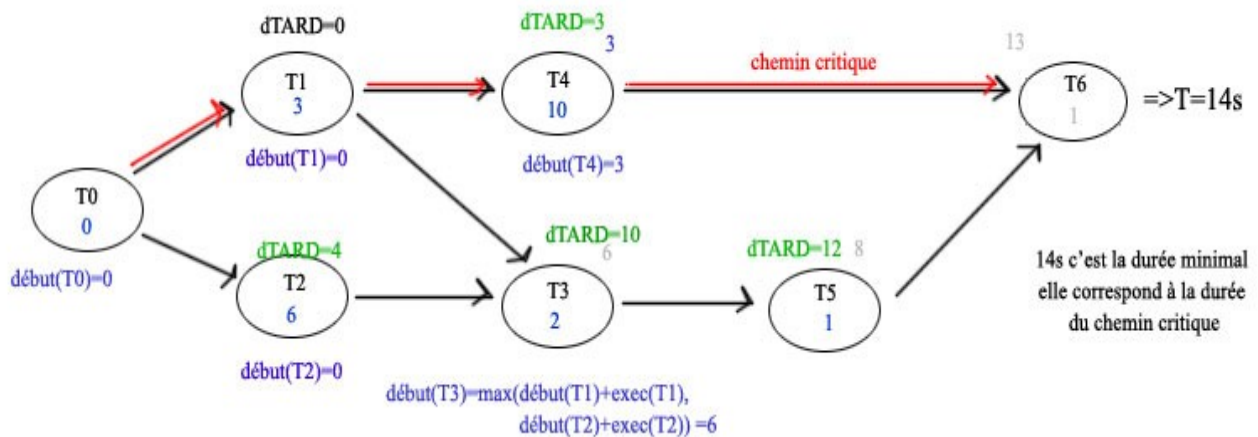
ORD \rightarrow (début(T0), proc(T))

Si $T1 \rightarrow T2$

Alors $\text{début}(T2) \geq \text{début}(T1) + \text{exec}(T1)$ ici $\text{exec}(T1)$ correspond à la durée de T1

Si $\text{début}(T1) \leq \text{début}(T2) \leq \text{début}(T1) + \text{exec}(T1)$

Alors $\text{proc}(T1) \neq \text{proc}(T2)$



On dira qu'une relation d'ordonnancement est minimale, si quelque soit le graphe proposé, la durée du chemin correspond à la durée du chemin critique

!/ Attention : minimal \neq optimal

- minimal : $T_n =$ chemin critique
- optimal : dans la configuration donnée T_n ne peut être plus petit

Problème : Il n'existe pas de d'algorithme exact d'ordonnancement minimal

Durée de table d'exécution d'un système de tâche : c'est le temps écoulé entre le début de T_0 et la fin de T_{n+1} , où n est le nombre de tâche dans le système.

Un ordonnancement O est minimal si quelque soit le nombre de processeurs, il n'existe pas d'autres ordonnancements dont la durée total d'exécution soit inférieure à celle de O.

Un ordonnancement O est optimal si pour la configuration processeur donnée, il n'existe pas d'autres ordonnancements dont la durée totale d'exécution soit inférieure à celle de O.

9.2 Cas d'une exécution statique (H1) sans communication (H4)

9.2.1 Cas ou l'on dispose d'assez de processeurs (H5)

Algo: * Calculer la date au plus tôt / au plus tard => algo de Bellman

* Affecter à chaque tâche une date de début (T1) comprise entre date_au_plus_tôt (T1) et date_au_plus_tard (Ti)*

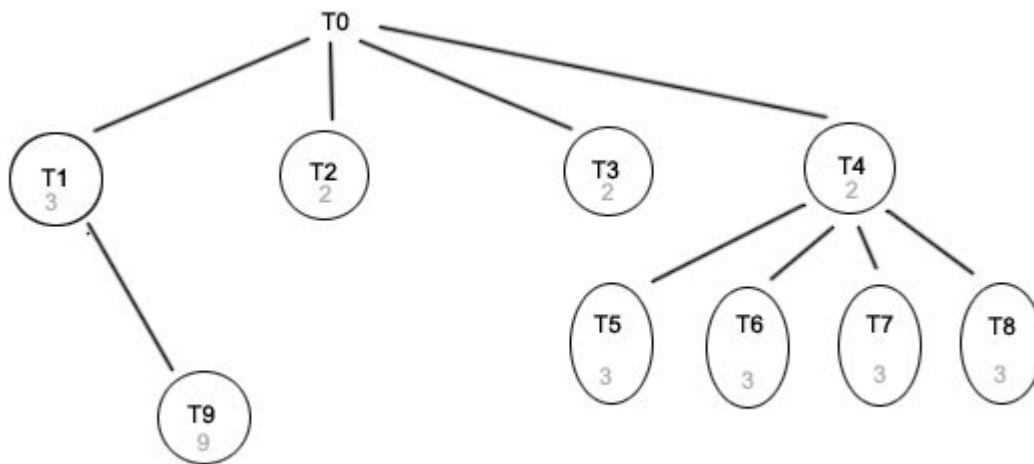
* Affecter a Ti un nouveau processeur

9.2.2 Cas avec pas assez de processeurs (non H5)

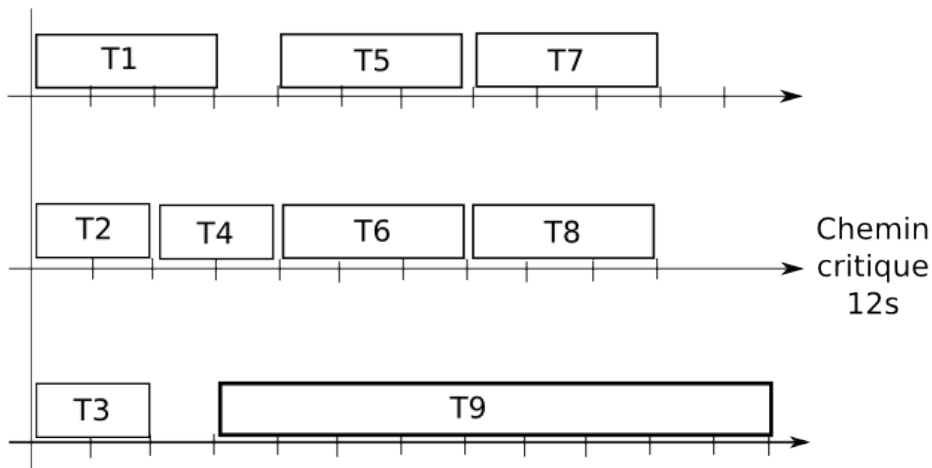
-> On calcule les dates au plus tôt / ou plus tard

-> on maintient une liste de tâches exécutables : - la tâche dont la date au plus tôt est passée
- la tâche dont tous les processeurs ont terminés

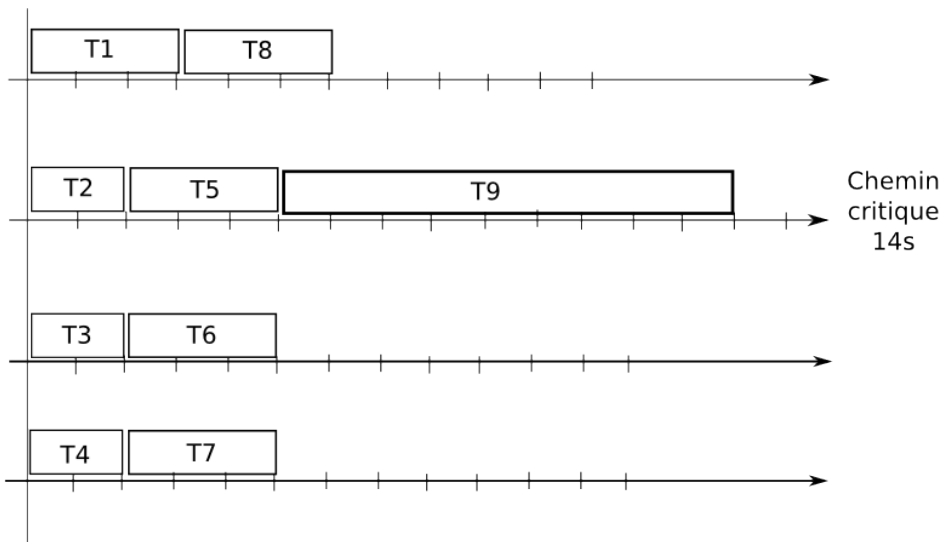
* Chaque fois qu'un processeur se libère, l'ordonnancement choisit une tâche



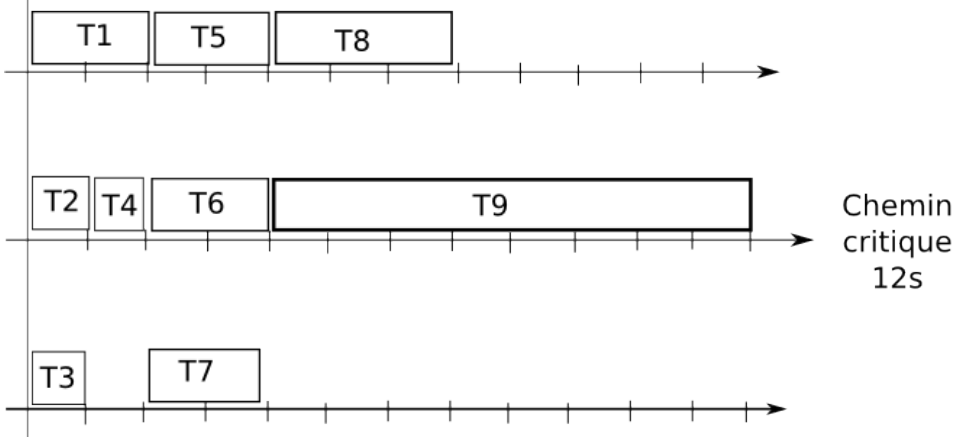
-> 3 processeurs : minimal 12s



-> 4 processeurs : minimal 14s



-> 3 processeurs : on diminue chacun d'une seconde

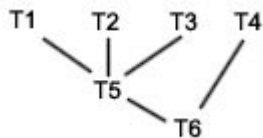


=> H1, H4, non-H5

* minimalité : il n'existe pas d'algorithme, que des euristiques

* optimalité :

- 1 processeur : OK
- 2 processeurs : OK
- p processeurs + "graphe" anti-arborescence : OK

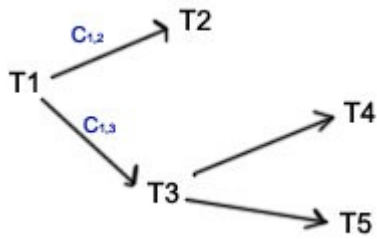


9.3 Exécution statique (H1) avec communication (non H4)

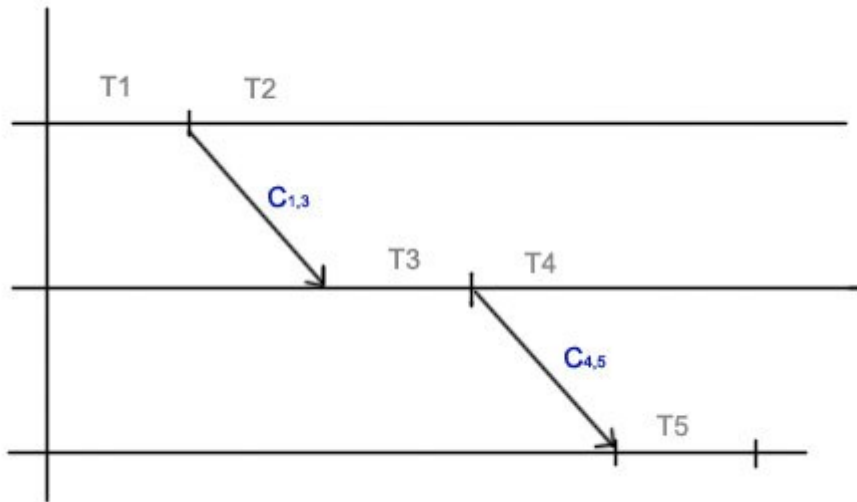
9.3.1 Communication et ordonnancement

On cherche la date au plus tôt : $date_au_plus_tôt (T_i) = \max (date_tot(T_j) + exec(T_j) + C_{i,j}) ?$

Idée : on considère que $C_{j,i} = 0$ si $proc(T_j) = proc(T_i)$

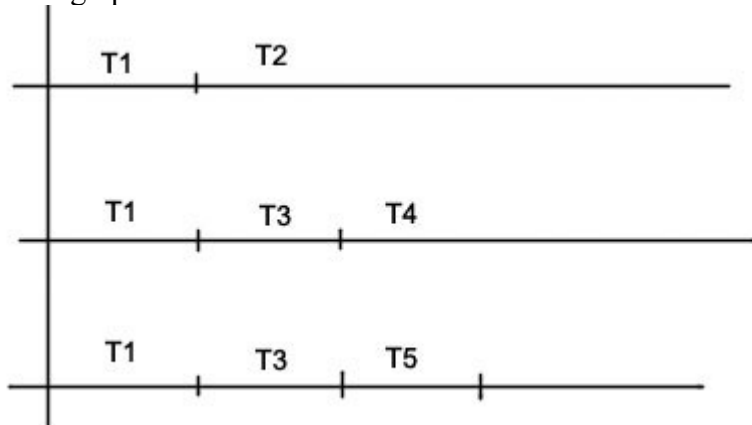


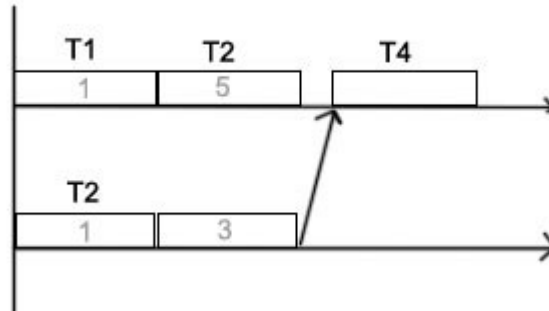
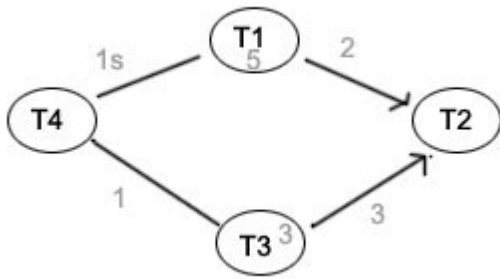
Les tâches sont duplicables :



H8 : les tâches sont duplicables

H9 : le graphe est un arbre => il existe un ordonnancement minimal





Cours 5/12/06

Schéma

T4 sur le même processeur que la tâche prédecesseur formant le chemin critique jusqu'à T4

- Pour tous les "prédecesseurs" T_j de T_i
- Calculer $Ch_j = date_tot(T_i) + ex(T_j) + C_{j,i}$
- Soit Ch_{jm} tel que Ch_i soit maximal
- Placer T_{jm} et T_i sur le même processeur
- $Ch_{jm} = date_tot(T_{jm}) + ex(T_{jm})$
- $date_tot(T_i) = \max(Ch_{jm}, Ch_{j \neq jm})$

Schéma

- T1 : $d(T1)=0$
- T2 : $d(T2)=0$
- T5 : $d(T5)=3$ $proc(T5) = proc(T2)$
- T3 : $d(T3)=4$ $proc(T3) = proc(T1)$
- T4 : | $Ch1=4+2$ | $proc(T4) = proc(T1)$
- | $Ch2=3+1$ | $d(T4) = \max(Ch1=4, Ch2=4) = 4$
- T7 : | $Ch2=4+2+1=7$ |
- | $Ch4=4+3+3=10$ | $d(T7) = \max(7,7,9)$
- | $Ch5=3+3+3=9$ |

Schéma

9.4 Cas où l'on ne dispose pas du graphe (non H1)

- HS1 : quel que soit le site où T_i s'exécute, il a toujours accès aux ressources dont il a besoin.
- HS2 : Un site est capable d'estimer sa charge.
- HS3 : Un site est capable d'évaluer les besoins d'un processus/tâche

9.4.3 Ordonnancement sans migration

Une tâche est affectée de façon définitive à un processeur.

- Solution centralisée
- Solution distribuée
 - > demande à l'ensemble des machines
 - > sondage
 - > sous-chaîne de machines

9.4.2 Ordonnancement avec migration

"Load balancing" équilibre des charges des processeurs.

Attention : Migrer un processus consiste à déplacer son code et données mais aussi son contexte (fichier ouvert, sémaphore...)

Appel d'offre / Offre de service
M1 surchargé / M1 sous-charge
demande l'aide/ demande du travail

9.5 Clusters / Grappes de processeurs

Station de travail S

Λ : débit et entrée de demande = 40 requêtes/s

M_{ju} : débit de sortie = 50rq/s

On peut montrer que temps d'attente moyen entre une demande et sa réponse : $T = 1 / M_{ju} - \Lambda$

$T_i = 1/50-40 = 1/10$ s

N stations du type S

$M_{ju} = M_{ju}$

mais $\Lambda = N \Lambda$

On repartit Λ sur les N stations

$\Lambda_i = 1/N(N.\Lambda) = \Lambda \quad \Rightarrow \quad T_i = 1 / M_{ju} - \Lambda$

On achète un multiproc à N processeurs $\Rightarrow M_{juS} = N.M_{ju}$

$T_{multi} = 1/ M_{juS} - \Lambda = 1/ N(M_{ju}-\Lambda)$