Programmation Concurrente

Systèmes Concurrents

- **cooperation:** activités *se connaissent*
- **competition :** activité *s'ignorent*

augmentation de la puissance de calcul + économies via mutualisation des donnés

-> services répartis + activités multi-processeur
=> archi multi-proc pour améliorer la puiss de calcul

Parallélisme: Exécution simultanée de plusieurs codes
Concurrence: Structuration d'un programme en activités ± indépendantes qui interagissent et se coordonnent.

	Pas de concurrence	**Concurrence**
Pas de parallélisme	prog séquentiel	multi activités sur un mono-proc
Parallélisme	parallélisation automatique / implicite	multi activités sur un mono-proc

Activités ± simultanées -> **explosion de l'espace d'état**
Interdépendance des activités -> **non déterministe**

Cohérence informatique/mémoire:

- **cohérence séquentielle:** résultat execution // est le même que celui d'une execution seq qui respecte l'ordre des proc
- **cohérence PRAM:** Les écritures d'un même processeur sont vues dans l'ordre où elles ont été effectuées: des écriture de processeurs différent peuvent être vues dans des ordres différents.
- **cohérence *lente*:** une lecture retourne une valeur précédemment écrite, sans remonter dans le temps.

Activité:

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

Activité communiquant par messages:

- communication par transfert de données (messages)
- coordination implicite (communication)
- designation nécessaire du destinataire (canal)

Contrôler:

- par progression et les interactions de chaque activité
- assurer leur protection réciproque
  => Attente par blocage suspension de l'activité

protocole -> séquences d'actions autorisée

Décrire:

- Compter les actions/changements d'états, les relier entre eux
- Triplet de Hoare: (précondition/action/postcondition)

p.35 -> 37 : Avantages/Inconvénients + performance

L'exclusion mutuelle (protocole d'isolation)

- **Section critique:** S_1, S_2 sections critiques qui doivent chacune être exécutées de manière atomique
  -> Résultat concurrente S_1 et S_2 même que une des exécutions seq $S_1; S_2$ ou $S_2; S_1$
  -> Contrôle l'ordre d'exécution de S_1 et S_2 (exclusion mutuelle) ou par effets de $S_1 S_2$ (contrôle de concurrence)
- **Prop:** au plus une activité en cours d'exécution d'une section critique (sûreté)
  si une demande $\rightarrow$ activité qui demande a entrer sera admise (progression)
  si activité demande à entrer, elle finira par entrer (vivacité individuelle)

Implémentation: `Test And Set`; `Fetch And Add`; Ordonnanceur avec priorités; système de fichiers
-> Utilisation de verrous avec méthodes acquire et release

Sémaphore

Gestion des interactions entre activités (isoler, synchroniser...)

- **Sémaphore:** encapsule un entier: $\geqslant 0$: opération `down` (décrémente le compteur bloque signal avant de débloquer) opération `up` (incrémente le compteur)

Pour eut Sémaphore: occurrence E := 0 -> signalement de la présence s.up() attendre et consommer s.down()

- **Sémaphore booléen:** verrou/lock
- **Allocateur de ressources:** N ressources, 2 opérations allouer et libérer -> sémaphore avec N jetons

Interblocage

- **Allocation de ressources multiples:** gérant -> demande + libère (rend réutilisable + libère à la terminaison)
- **Correction:**
  - sûreté : rien de mauvais ne se produit (exclusion mutuelle, invariants du programme)
  - vivacité: qq ch de bon finit par se produire (équité, absence de famine, terminaison de boucle) -> p.8
- **Famine:** Une activité est en famine lorsqu'elle attend infiniment longtemps la satisfaction de sa requête (elle n'est jamais satisfaite)
- **Interblocage:** Allocation de ressources réutilisables, non réquisitionnables, non partageables, en quantité entières et finies, dont l'usage est indépendant de l'ordre d'allocation. (entrelacement peut entraîner de l'interblocage..)
- **Def:** Un ensemble d'activités est en interblocage (dead lock) <=> toute activité de l'ensemble est en attente d'une ressource qui ne peut être libérée que par une autre activité de l'ensemble.

Absence de famine => Absence d'interblocage
L'interblocage est un état stable.

Prévention: empêcher la formation de cycles

Détection + guérison: le détecter et l'éliminer

- *Eviter l'accès exclusif*
- *Eviter la redemande bloquante*
- *Eviter l'attente circulaire*

Moniteur

module exportant des procédures + contraintes d'exclusion mutuelle + synchro interne

- **Variable condition:** wait (bloque activité libère accès exclu au moniteur)
  signal (si activités bloquées sur C elle en débloque une)

Code exécuté en exclu mutuelle

Méthodologie:

Déterminer l'interface du moniteur

Énoncer les prédicats d'acceptation de chaque opération

Déduire es variables d'états -> écrire les prédicats

Formuler l'invariant du moniteur et les prédicats d'acceptation

Pour chaque prédicat, définir une variable condition

Programmer

Rendez-vous **Adapté à la répartition $\neq$ semaphore -> moniteur -> centralisé**

Comme modèle client/serveur

Tâche:

- activité
- demander un RDV avec une autre tâche
- attendre un RDV soit un/plusieurs point(s) d'entrée