Introduction aux systèmes temps-réel et aux architectures multicoeurs

Fabien Calcado

Email: *fabien.calcado@gmail.com*

Contenu du cours n°1

- **Rappels**
  - Généralités sur le développement logiciel
  - Système multitâche
- **Gestion du parallélisme (rappel ?)**
  - Communication et contrôle de la concurrence
    - Mutex / sémaphore

Plan du cours

- **Généralités**
- **Système multitâche**
- **Gestion du parallélisme**

Généralités

- **Un système de traitement de l'information est appelé à remplir une mission :**
  - Mise en œuvre d'un certain nombre de *fonctions*
  - *Suites* d'instruction codées (programme)
  - Des informations *codées* et *organisées*
- **Réalisation**
  - Nécessite la mise en œuvre de ressources matérielles
    - HW => relativement figées
  - Logiciels
    - SW spécifiques

Généralités

- **Qualités des logiciels**
  - Efficaces
    - **réaliser les fonctions requises avec des performances adaptées**
  - Fiables
    - **Corrects, complets, sûrs**
  - Testables
    - **compréhensibles, lisibles, structurées, auto-descriptifs**
  - Portables
    - **sur différentes machines**
  - Maintenables
    - **corrections**
  - Réutilisables
    - **évolutions**
  - Certifiables
    - **prouver le bon comportement**

Généralités

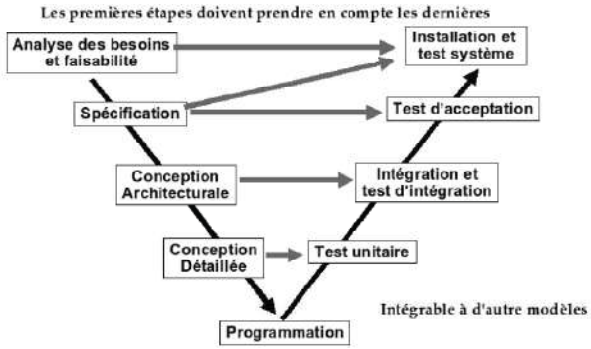
- **Cela implique:**
  - Une conception conforme à la mission du système
  - Une réalisation conforme à la conception
- **Faire une analyse des spécifications:**
  - Des méthodes et des règles
  - Des techniques élémentaires
  - Conception / programmation / mise en oeuvre

Généralités

- **L'approche modulaire:**
  - Permet la *division* de la complexité d'un problème
  - Le *partage du travail* à réaliser
- **Les méthodes pour la cohérence des modules:**
  - Cohérence logique
    - répartition par classe de problème, matériel ou logiciel
  - Cohérence temporelle
    - synchronisation, enchaînement des traitements
  - Cohérence procédurale
    - organisation des algorithmes
  - Cohérence par données communes
    - approche objet
  - Cohérence fonctionnelle
    - 1 fonctionnalité donne 1 module

Généralités

- **Cycle de conception en V:**



Généralités



- **Le modèle en V:**
  - Analyse des besoins
    - **Experts du domaine d'application**
    - **Environnement, rôle, ressources, contraintes**
      - » Que veut-on ? A quel coût ?
  - Spécification globale (fonctionnelle)
    - **Ensemble d'exigences**
    - **Description de l'action du logiciel, pas de décision sur la réalisation**
      - » Réponse attendue par le système lorsqu'il est soumis à une sollicitation particulière ➔ Ce qu'il doit faire ?
  - Conception architecturale (et détaillée)
    - **Décomposition du logiciel, spécification des interfaces, description de la réalisation des composants**
      - » Comment il va le faire ?

Généralités



- **Le modèle en V:**
  - Programmation
    - Phase de réalisation
      - » Loin d'être la partie la plus importante
  - Test unitaire
    - S'assurer du fonctionnement correct d'une brique
      - » Respect du cahier des charges de manière individuelle
  - Intégration et test d'intégration
    - Regroupe les briques pour tester le produit

Généralités



- **Le modèle de conception en V**
  - Vérification et validation (logicielle)
    - Adéquation aux besoins, satisfaction de la spécification
    - Analyses, tests
    - Les erreurs logicielles
      - » Plus de la moitié des erreurs sont dues une mauvaise conception
      - » Plus de la moitié des erreurs sont détectées chez les clients
    - Coût
      - » Le logiciel représente la majeure partie du coût
      - » ~ 40 à 60% des dépenses sont consacrées aux tests et aux corrections du logiciel !

Généralités



- **Quelques exemples de bug**
  - En 1996, la fusée Ariane 5 a explosé en vol
    - Système de navigation identique que sur Ariane 4 mais non testé sur Ariane 5…
      - » économies de 800 000 frs sur le coût des préparatifs
  - En 2000, en médecine, un programme de mesure de radiation a donné des valeurs incorrectes
    - A couté la vie à 8 patients et ~20 personnes grièvement blessées
  - En 2009, quelques dizaines de milliers de comptes de clients BNP Paribas ont été débités par erreur

Plan du cours

- Généralités
- **Système multitâche**
- Gestion du parallélisme

Programmation multitâche

- **Système**
  - Dispose :
    - Plusieurs ressources
      - » CPU(s), mémoire, disques durs, cartes réseau …
    - Chacune capable d'assurer une fonction à la fois
  - Afin de remplir :
    - Plusieurs fonctionnalités
    - Application = une ou plusieurs tâches (une ou plusieurs fonctions)
      - » Indépendantes ou non
      - » Rythmes d'activation différents ou pas entièrement définis

➔**Besoin de partager les ressources entre différentes tâches afin de profiter du parallélisme**

Programmation multitâche

- **Système**
  - Architecture logicielle
    - Ensemble de tâches (programmes) à exécuter de manière concurrente
  - Architecture matérielle
    - Ensemble limité de ressources de traitements (CPUs) interconnectées
      - » Architecture mono / multiprocesseur
      - » Architecture répartie ou distribuée

Programmation multitâche

- **Mise en œuvre d'un système**
  - Cela suppose une allocation des tâches sur les diverses ressources au cours du temps
    - Faire cette allocation au cours du temps c'est ordonnancer les tâches
    - Contexte temps réel ➔ l'ordonnancement doit satisfaire les contraintes temporelles de l'ensembles des tâches
  - Terminologie
    - L'**ordonnancement** est l'organisation de l'exécution des tâches sur les ressources du système
      - » Séquencement , entrelacement…
    - La **politique d'ordonnancement** est la règle d'organisation de l'exécution des tâches



Monoprogramming

- Un ordinateur :
  - 1 processeur, de la mémoire et d'autres ressources
- Tout faire en une tâche (programmation en boucle)
  - Programmation cyclique : activation périodique de la tâche
- Avantage :
  - Simple à mettre en œuvre
  - Vérification simple (déterministe)
- Inconvénients :
  - Conception longue et pénible
  - Mauvaise utilisation des ressources
  - Évolution difficile



Multiprogramming

- Un programme a besoin
  - D'un ou plusieurs processeur virtuel (process/thread)
  - D'une mémoire virtuelle (espace d'adressage)
  - De ressources virtuelles
  - Exemple du processus UNIX
    - » 1 processus = 1 « processeur virtuel »
- Faire fonctionner plusieurs programmes en parallèle
- L'OS est le programme en charge de la multi-programmation
  - Isolation entre les programmes
  - Partage des ressources



Isolation des programmes

- Se prémunir d'éventuel programme défaillant
  - Isolation accès mémoire (MMU)
  - Accès aux ressources sécurisé
    - » Services système (noyau)
    - » Assure une utilisation correcte des ressources
  - Programmation défensive
    - » Vérifications de viol de quotas, hypothèses sur les interruptions…
- Sûreté et sécurité ➔ conception similaire
- Sûreté de fonctionnement ≠ zéro défaut



Partage des ressources

- Allocation « spatiale »
  - Possible si plusieurs ressources
    - » CPUs, Mémoire…
- Allocation « temporelle »
  - Au cours du temps
  - Obligatoire si une seule ressource
    - » D'un CPU, accès du disque, accès du port série…

➔ **Ordonnancement CPU(s) : organisation de l'exécution des tâches par le(s) processeur(s) du système**

Programmation multitâche

- **Besoins primordiaux**
  - La communication entre les tâches
    - Transfert de données
  - La synchronisation entre les tâches
    - ordonnancement des traitements les uns par rapport aux autres
- **Propriétés souvent assurées**
  - La cohérence des données
  - Le déterminisme de l'exécution
    - indépendant du parallélisme

Programmation multitâche

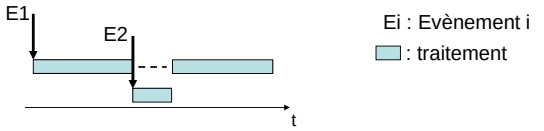
- **Sources d'incohérences**
  - Ne sont pas dû au parallélisme…
  - … mais aux interactions entre des programmes se déroulant en parallèle
    - Mémoire commune
    - Ressources partagées
    - Communications (ordres des)…

Programmation multitâche

- **Modèles d'interaction avec le monde extérieur**
  - Scrutation cyclique (polling)
    - Interroger régulièrement le ou les périphérique(s)
    - Mise en œuvre sous forme de boucle infinie
    - Avantage
      - » simplicité de mise en œuvre
    - Inconvénients
      - » ingérable avec un grand nombre de périphériques
      - » Données non disponibles ➔ perte de temps à interroger inutilement les pilotes du périphérique

Programmation multitâche

- **Modèles d'interaction avec le monde extérieur**
  - L'interaction par interruptions
    - Évènement qui provoque un changement dans l'exécution normale d'un programme
      - » Gérer les différentes échelles de temps
      - » Interruption d'entrée / sortie, d'horloge, etc (causes externes)
    - Illustration
      - » traitement lié à E2 prioritaire par rapport à E1

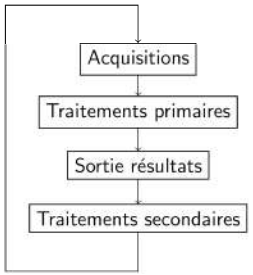


- **Modèles d'interaction avec le monde extérieur**
  - L'interaction par interruptions
    - Avantages
      - » Grande souplesse
      - » Relativement simple
      - » Optimisation possible
    - Inconvénients
      - » Cohérence des données (entrelacement)
      - » Pertinence (violation d'une contrainte essentielle)
      - » Partage des ressources (problème de deadlock / famine)

- **Modèles d'interaction**
  - L'interaction par interruptions
    - Peut désigner également les exceptions ou déroutements
      - » causes internes au programme
      - » Exemple : instruction erronée, accès à une zone mémoire inexistante, calcul arithmétique incorrect,…
      - » *Attention aux architectures à exécution dans le désordre (OoO)*

- **Modèles d'interaction**
  - Cas du système multi-tâches
    - Plusieurs tâches (programmes ou suites d'instructions)
    - Commutation de tâches
      - » Interrompre une tâche (par exemple en attente) pour exécuter une autre tâche
      - » Interruption (périodique) déclenchée par une horloge

- **La programmation en boucle**
  - Permet d'éviter les problèmes liés au multi-tâches
    - Flot de contrôle statique
    - Pas de préemption



- **La programmation en boucle**
  - Avantages
    - « Très simple »
    - Garanties par le temps de cycle
  - Inconvénients
    - Rigide
    - Pas optimal
    - Peu performant
  - Exemple : trois tâches A, B, C
    - A peut être décomposé en A1 et A2
    - C peut être décomposé en C1, C2 et C3
      - » Problème de dépendance vis-à-vis de la vitesse du processeur !

- **Composition parallèle de programme**

$$P = P_1 * P_2$$

  - Connaissant P_1 et P_2, que peut-on dire de P ?
  - Caractériser les interactions, explicites ou implicites
    - Cas asynchrone: produit possible ou non
    - Cas synchrone: produit synchronisé d'automate
  - La programmation en boucle permet une composition rigide et simple, mais peu efficace
  - Une interruption peut introduire une désynchronisation

- Généralités
- Système multitâche
- **Gestion du parallélisme**

- **Exemple de mise à jour de comptes bancaires:**

$val : ENTIER$

```
PROCESSUS Crediteur(c : ENTIER){
    val ← val + c   (4)
}
```

```
PROCESSUS Debiteur(d : ENTIER){
(1) si val < d alors
(2)    Ecrire("decouvert")
    finsi
(3)    val ← val − d
    }
```

  - Problème si on exécute:
    - Val = 5, debiteur(6), crediteur(4)
    - Séquence: (1), (4), (2), (3)... découvert signalé ! (val = 3)
  - Problème si on exécute:
    - Val = 5, debiteur(4) en // debiteur(3)
    - Séquence: (1), (1), (3), (3) …. aucun découvert signalé ! (val = -2)



- **Interactions entre les programmes (rappel)**
  - Les problèmes dans les systèmes multi-tâches ne sont pas dû au parallélisme mais aux interactions entre les tâches se déroulant en parallèle
  - Partage des ressources
    - Garantir que l'exécution en parallèle de plusieurs tâches fournit le même résultat qu'une exécution strictement séquentielle
  - Communication, coopération
    - Garantir que l'échange d'information entre les programmes obéit à un protocole défini



- **Pourquoi synchroniser ?**
  - Résoudre les problèmes de cohérence mémoire sur la communication de données (en mémoire partagée)
  - Spécifier les dépendances entre les traitements
    - Contrôler l'ordre d'exécution des tâches
      - » Ex : producteur / consommateur (permet à un thread de contrôler quand un autre s'exécute)
      - » Ex: commandes de périphérique / du matériel (s'assurer qu'on envoie pas deux commandes disques contraires)
  - En général : régler les problèmes de concurrence sur l'accès à une ressource partagée
    - logicielle ou matérielle



- **Mécanismes de communication**
  - Mémoire partagée, tubes FIFO, boites aux lettres asynchrones, buffer circulaires…
  - Toute communication entre deux tâches utilise à un moment de la mémoire partagée
    - Éventuellement cachée par le noyau
      - » Mécanisme principal à maîtriser
  - Attention aux problèmes « cachés »
    - Une instructions C = plusieurs instructions assembleur !
      - » Exemple cours n°1: une variable, deux tâches
        - ➔ l'une incrémente, l'autre décrémente



- **Définition : section critique**
  - Portion de code dans laquelle une tâche utilise des ressources qui peuvent être utilisées par d'autres tâches, mais qui ne peuvent pas être utilisées par celles-ci au même moment
    - Un exemple commun de ressource partagée est un ensemble de blocs mémoire
    - Permet de s'assurer qu'une portion de code est exécutée de manière séquentielle
  - Attention, les sections critiques diminuent le taux de parallélisme
    - il faut donc (essayer de) les minimiser

Gestion du parallélisme

- **Définitions de sériabilité et d'atomicité**
  - A et B sont deux tâches (traitements)
  - Sériabilité
    - A // B indépendant de l'ordonnancement
    - A // B = A,B = B,A
  - A est atomique par rapport à B si
    - A ne peut pas être mis en // avec B
    - A ne peut pas être préempté au profit de B
    - B ne peut pas observer d'états intermédiaires dans A pendant l'exécution de A
    - A dure un temps nul pour B

Gestion du parallélisme

- **Remarques**
  - Les périodes d'atomicité diminuent le taux de parallélisme
    - Elles ne doivent pas faire rater les échéances
    - Elles doivent être brèves en multiprocesseur
  - L'atomicité évite certaines interactions
    - Ne résout pas A,B = B,A
    - Exemple : Décodage parallèle de code Morse

Gestion du parallélisme

- **Remarques**
  - Exemple : codage parallèle de code Morse
    - Chaîne à coder « SOS »
    - « S » = « … » , « O » = « - - - »
    - Deux threads, l'un code les « S » l'autre les « O »
    - Sans précautions : « ….-.--. »
    - Respect de l'atomicité : « ……--- »
    - Respect de l'ordre: « …---… »

    ➔ Section critique (mutex) résout l'atomicité mais pas les problèmes d'ordre

Gestion du parallélisme

- **Attitude face aux risques d'incohérence**
  - A et B doivent être atomiques l'un vis-à-vis de l'autre
    - Synchronisation pessimiste : Prévention (section critique)
      - » Atomicité : on empêche le problème de survenir
    - Synchronisation optimiste : Guérison (estampilles)
      - » On détecte le problème (données incohérente ➔ cohérentes)
    - Dépend de la possibilité (probabilité) d'avoir A et B actifs ensembles ?
      - » estampilles : peuvent ne pas finir

Gestion du parallélisme

- **Attitude face aux risques d'incohérence**
  - Guérir :
    - Copier la date *(l'estampille)*
    - Copier les données
    - Calculer les nouvelles valeurs

      ** Début atomicité **
    - Copier la date actuelle
      - » Est-ce que quelqu'un a rechangé l'estampille ?
    - Si elle est inchangée

      ➔ modifier les données et augmenter la date

      ** Fin atomicité **

      Sinon recommencer

Gestion du parallélisme

- **Une solution pour le problème de l'exclusion mutuelle vérifie les propriétés suivante:**
  - Indépendante de la vitesse d'exécution des programmes
  - Deux processus (ou plus) ne peuvent se trouver simultanément en section critique
  - Un processus hors de sa section critique et qui ne demande pas à y entrer ne doit empêcher un autre processus d'entrer en section critique
  - Deux processus ne doivent pas s'empêcher mutuellement et indéfiniment d'entrer en section critique
    - Phénomène d'interblocage
  - Un processus doit toujours entrer en section critique au bout d'un temps fini
    - Phénomène de famine

Gestion du parallélisme

- **Sémaphore:**
  - Un sémaphore est un objet sur lequel seulement 2 opérations atomiques sont possibles
    - P(sem) : décrémentation de la valeur du sémaphore « sem »
      - » ***Blocage si la valeur est < 0 (barrière)***
    - V(sem) : incrémentation de la valeur du sémaphore « sem »
      - » ***Permet de débloquer un processus bloqué par P (laissez-passer)***

Note : vient du Hollandais Passeren (prendre) , Vrygeven (libérer)

Gestion du parallélisme

- **Sémaphore d'exclusion mutuelle (mutex):**
  - Sémaphore binaire initialisé à 1
  - Sert à protéger une section critique
  - Permet d'assurer l'accès à des ensembles disjoints de variables partagées
    - Associer un sémaphore d'exclusion mutuelle à chaque ensemble

$mutex1, mutex2 : \mathrm{INIT}(TRUE)$

$\mathbf{P}(\mathrm{mutex1})$
...
$\{section\ critique\ n°1\}$
...
$\mathbf{V}(\mathrm{mutex1})$
...
$\mathbf{P}(\mathrm{mutex2})$
...
$\{section\ critique\ n°2\}$
...
$\mathbf{V}(\mathrm{mutex2})$

- **Sémaphore privé**
  - Quand chaque tâche n'est autorisée qu'à faire une seule des primitives P ou V
    - Le sémaphore est dit privé (emploi particulier)
  - Interprétation
    - Le(s) processus qui fait P attend un signal du(des) processus qui fait V
  - Propriétés
    - Si le processus récepteur est en avance il est bloqué
    - Si le signal est émis en avance il est mémorisé

- **Exemple d'utilisation du sémaphore privé**
  - Un processus doit être activé par un autre sur un évènement
    - Un seul processus peut exécuter l'opération P
    - Les autres processus peuvent exécuter l'opération V

Processus réveilleur 1	Processus réveilleur 2	Processus à activer
...	...	...
V(sp)	V(sp)	P(sp)
...	...	...

- **Problème avec les sémaphores**
  - Deux tâches A et B et deux sémaphores S1 et S2 avec M(S1) = M(S2) = 1
  - La chronologie suivante
    - A : P(S1)
    - B : P(S2)
    - A : P(S2) /* A est bloquée dans P */
    - B : P(S1) /* B est bloquée dans P */
  - Remarque
    - C'est un problème global
      - » A est bloquée et c'est B qui peut faire évoluer la situation
      - » B est bloquée et c'est A qui peut faire évoluer la situation
      - ➔ La situation ne peut plus évoluer
    - On peut s'interbloquer avec un seul sémaphore (interrupt handler =>TP n°2)

- **Solutions**
  - Remède
    - Annuler l'un des appels à P
    - Il faut faire remonter la tâche dans son code
    - Il faut pouvoir restaurer les données de la tâche
    - Annuler toutes les opérations qu'elle a faite
    - En pratique : détection, et destruction des tâches concernées
    - Même problème qu'avec les estampilles…
  - Prévention
    - Problème compliqué mais dans des cas particuliers, il y a des solutions simples
    - Annonces des besoins
      - » La tâche demande en une seule fois toutes les ressources dont elle a besoin

Gestion du parallélisme

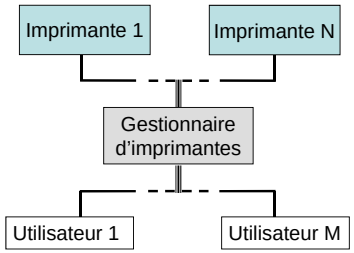
- **Ressources ordonnées partiellement**
  - La tâche peut faire des demandes successives pourvu qu'elles portent sur des ressources comparables
  - Qu'elles soient faites en suivant une **relation d'ordre**
  - Démonstration
    - La tâche qui occupe la ressource de plus grand rang ne peut être bloquée
    - La condition n'est pas nécessaire
      - » A : P(m1) , P(m2) , P(m3)
      - » B : P(m1) , P(m3) , P(m2)

Gestion du parallélisme

- **Sémaphores généraux:**
  - Utilisés comme compteurs de ressources
    - N'est pas limité à 0 ou 1 contrairement au sémaphore d'exclusion mutuelle
  - Valeurs du sémaphore
    - Initiale représente le nombre de ressources maximum
    - Courante le nombre de ressources disponibles
  - L'opération P permet d'acquérir une ressource
    - Bloqué si aucune ressource n'est disponible
  - L'opération V permet de libérer une ressource
    - Signale la disponibilité d'une ressource et débloque un processus éventuellement en attente

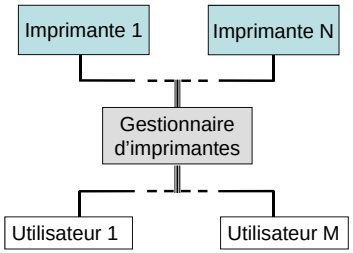
Gestion du parallélisme

- **Exemple du Pool d'imprimantes:**
  - Valeur initiale du sémaphore (?)
  - Les M processus utilisateurs font (?)
  - Le gestionnaire fait (?)



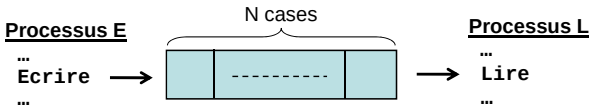
Gestion du parallélisme

- **Exemple du Pool d'imprimantes:**
  - Valeur initiale du sémaphore ➔ N (nombre de ressources)
  - Les M processus utilisateurs font des P( )
  - Le gestionnaire fait des V( ) pour libérer une ressource

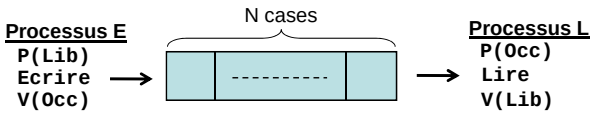


- **Producteur / Consommateur:**
  - Le système possède N emplacements pour stocker de l'information
    - Les processus ***producteurs*** produisent de l'information vers ces emplacements
    - Les processus ***consommateurs*** utilisent l'information et libère l'emplacement correspondant
  - Un sémaphore est nécessaire pour synchroniser les deux types de processus
    - Bloquer un producteur s'il n'y a plus de place
    - Bloquer un consommateur s'il n'y a plus d'information disponible

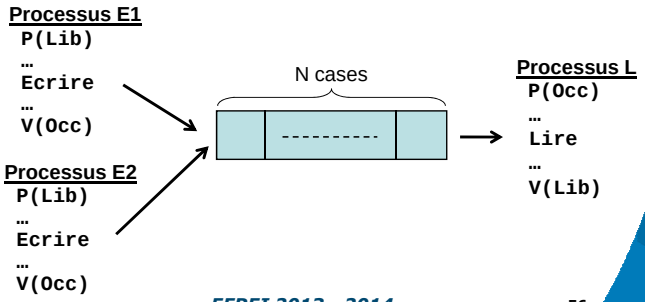
- **Exemple d'un tampon de Lecture / Ecriture :**
  - Processus E écrit des données dans le buffer
  - Processus L lit les données dans le buffer
  - Valeurs initiales des sémaphores ?



- **Exemple d'un tampon de Lecture / Ecriture :**
  - Processus E écrit des données dans le buffer
  - Processus L lit les données dans le buffer
  - Valeurs initiales des sémaphores ➔ Lib=N, Occ=0



- **Exemple d'un tampon de Lecture / Ecriture**
  - Cas avec plusieurs Ecrivains et 1 seul Lecteurs
    - Quel type de problème survient ?





- **Exemple d'un tampon de Lecture / Ecriture**
  - Cas avec plusieurs Ecrivains et 1 seul Lecteurs
    - ➔ Problème d'exclusion mutuelle