Conception et Validation des Systèmes Temps-Réel : Architecture, Parallélisme et Sûreté de Fonctionnement

V. David
Expert Senior IRSN
Expert Systèmes Temps-Réel, Sûreté de Fonctionnement
Précédemment Krono-Safe, Directeur Technique
Fondateur du LaSTRE du CEA LIST
Pr. INSTN

Systèmes Asynchrones

- **Introduction**
- **Cohérence globale**
- **Atomicité du matériel**
- **Atomicité du logiciel**
- **Sémaphores**
- **Famine, Inter-blocage**
- ***Réseaux de Petri***
  - ***Modélisation***
  - ***Analyse***

Introduction

- **Objectifs**
  - **produire des résultats (justes) à des dates données**
  - **gérer des échelles de temps différentes**
- **Expression du temps**
  - **en dehors de la machine de Turing**
  - **par un périphérique**
    - » **compteur programmable**
    - » **décrémentation périodique**
    - » **interruption au passage par zéro**
- **Exemple**
  - **voiture**
    - » **commande allumage moteur**
    - » **commande alimentation moteur**
    - » **commande de suspension**
    - » **ABS, ASR, ESP…**
    - » **IHM**

Programmation Multitâches

- **Objectifs temps réel**
  - imposée par la cohabitation de périodes différentes
- **Objectif général**
  - optimisation de l'emploi des ressources

- **Rôle du moniteur :**
- **interface avec le matériel**
  - l'initialisation
    - » de la mémoire physique
    - » des registres
    - » du dispositif de mémoire virtuelle
  - gestion du boîtier horloge
    - » gestion du temps
    - » partage de l'horloge entre les traitements
- **ordonnancement**
  - participe au respect des échéances
  - en déchargeant la conception du découpage
  - en tranche des traitements longs
- **communication entre traitements**
  - transport d'information d'un traitement à un autre
  - soit édition de liens séparée
  - soit protection mémoire
- **cohabitation d'applications devant s'ignorer**
  - exécution comme seul sur le calculateur
  - partage du CPU*
  - partage du disque
  - …

- **Les briques de base**
  - **I/O avec événement**
  - **multiprogrammation**
  - **synchronisation**
  - **communication**
  - **I/O anticipées ou différées**
  - **ordonnancement HPF**
    - » **Avec révision des priorités**
  - **gestion du temps**
    - » **heure courante**
    - » **chien de garde**
    - » **sommeil, réveil**

- **Pourquoi des moniteurs <u>temps réel</u> ?**

  (le suivi d'un glacier est possible avec tout calculateur)

  - **vitesse**
  - **précision du temps**

Cohérence globale

- **Une liste chaînée d'éléments en mémoire partagée :**
  - **debut référence le premier élément**
  - **fin référence le dernier élément**
  - **debut et fin valent 0 si la liste est vide**
  - **2 opérations sur la liste:**
    - » **ajouter() et rechercher()**
- **Deux processus :**
  - **L'un veut ajouter**
  - **L'autre veut rechercher**

```
ajouter(E *elem)
{p1 : if (fin NE 0)
 p2 :  fin->suivant= elem;
       else
 p3 :  debut= elem;
 p4 : fin= elem;
 p5 : fin->suivant= 0;
}

rechercher(E *elem)
{c1 : E *e= debut;
 c2 : for ( ; e NE 0 ;
 c5 :  e= e->suivant )
 c3 :if (test(e, elem))
 c4 :           break;
 c6 : return e;
}
```

- **sources d'incohérences :**
  - **ce n'est pas le parallélisme**
  - **mais les interactions entre programmes se déroulant en parallèle**
    - » **mémoire commune**
    - » **ressources partagées**
    - » **communications (ordre des)**
    - » **...**

- **Les hypothèses conditionnent la validité des démonstration sur les programmes**
  - **les hypothèses sont-elles toujours valides ?**
  - **attention aux préemptions, aux multicoeurs, aux caches non partagés : non von Neumann… raisonnement différent**

- **Aspect lié aux interruptions**
  - **un traitement "p" est interrompu à la date "t" pour faire un traitement "u" puis est repris**
- **Expérience :**
  - **Test : p1, u, p2**
  - **Test' : p1', u, p2'**
- **| t-t'| < précision des mesures de temps**

- **Problème :**

  les résultats des suites d'instructions

  p1, u, p2

  et

  p'1, u, p'2

  sont-ils identiques ?

- **Remarques :**
  - **"p" et "u" dans les 2 tests se terminent (probablement) aux mêmes dates relatives**

Définitions

- **Sériabilité :**
  - **A // B est indépendant de l'ordonnancement**
  - **A // B = A , B = B , A**
- **Atomicité :**

  A est atomique par rapport à B si
  - **A ne peut pas être mis en // avec B**

    ou
  - **A ne peut pas être préempté au profit de B**

    ou
  - **B ne peut pas observer d'états intermédiaires dans A pendant l'exécution de A**

    ou
  - **A dure un temps nul pour B**

- **Remarques :**
  - **les périodes d'atomicité diminuent le taux de // isme**

    ==>
    - » **elles doivent être brèves en multicpu**
    - » **elles ne doivent pas faire rater les échéances**
  - **l'atomicité évite certaines interactions**
    - » **mais ne résout pas A, B = B, A**

- **Attitude face aux risques d'incohérence :**
  - **A et B doivent être atomiques l'un vis à vis de l'autre**
    - » **prévention ==> atomicité**
    - » **guérison ==> estampilles (formalisé par Lamport)**
  - ***Possibilité (probabilité) d'avoir A et B actifs ensemble ?***

- **Guérir :**
  - **copier la date (l'estampille)**
  - **copier les données**
  - **calculer les nouvelles valeurs**
    - ***début atomicité***
  - **copier la date actuelle**
  - **si elle est inchangée**
    - » **modifier les données**
    - » **et**
    - » **augmenter la date**
      - ***fin atomicité***
  - **si non recommencer**
- **Problèmes :**
  - **si les données sont partagées et bougent :**
    - » **faire un calcul en temps borné**
    - » **ne pas dérouter**

Atomicité du matériel

- **Mono-cpu sans interruptions**
  - **pour quitter un programme A il faut que :**
    - » **A déroute ou**
    - » **A appelle un autre programme**
    - » **A fasse "return"**
    - – ***A ne peut pas être préempté à son insu***

- **Monocpu avec interruption**
  - **quand une interruption prioritaire se présente**
  - **le processeur peut interrompre le programme en cours A**
  - **au profit du programme B associé à l'interruption**
  - **l'interruption n'a lieu qu'à la fin de l'instruction en cours**
  - **Prévention par le masquage d'interruption**

- **Multicpu à bus unique**
  - **ici A et B sont les micro-codes des instructions**
    - » **la synchronisation est assurée par l'arbitre de bus**
    - » **l'accès élémentaire à la mémoire est atomique (lire ou écrire un mot sur le bus)**
  - **si l'interface entre le bus et le cpu le permet, le bus peut être monopolisé pour plus d'une opération**

    ==> l'instruction de type "TAS" peut être implémentée
    - » **atomicité au niveau des programmes, réalisée par consensus :**
    - » ***sur le cpu A ou B :***
      - `while(tas(verrou)) {/* attendre */ ;}`
      - `{corps de A ou B}`
      - `verrou= 0; /* libération */`

- **Multicpu à mémoire commune sans instruction TAS :**
  - **Solution compliquée [CROCUS]**

Architecture multiCPU à bus unique

- **L'occupation d'une ressource est un problème équivalent à l'atomicité**

- **Contraintes à respecter :**
  - **a) à tout instant, au plus un processus occupe une ressource**
  - **b) si plusieurs processus attendent la ressource et si elle est libre, alors l'un des processus doit pouvoir la prendre au bout d'un temps fini**
  - **c) si un processus ne tente jamais d'occuper la ressource (ou s'arrête en ne possédant pas la ressource), il ne doit pas bloquer les autres processus**
  - **d) la solution doit être la même pour tout les processus (aucun processus n'a de rôle particulier : solution symétrique)**

- **Construction de la solution par étapes :**
  - **avec 2 processus**
  - **une solution existe pour un nombre connu de processus**

Solution partielle 1

- **Les processus partagent une variable nommée R :**
  - **R == 0 signifie que la ressource est libre**
  - **R == 1 signifie que la ressource est occupée**
  - **R est initialisée à 0**

- **Programme 1 :**

```
L1: while (R == 1) {;} /* sortie quand R == 0 */
L2: R= 1;
    /* la ressource est occupée */
L3: R= 0;
    /* la ressource est libérée */
```

b) c) d) sont satisfaites

a) ne l'est pas

Contre-exemple :

A:L1, B:L1, A:L2, B:L2

Solution partielle 2

- **Les processus partagent une variable nommée T :**
  - **T == i signifie que le processus i a le droit d'occuper la ressource**
  - **T != i signifie que le processus i n'a pas le droit d'occuper la ressource**
  - **T est initialisée indifféremment à 1 ou 2**

- **Programme 2 :**

```
L1: while (T != moi) {;} /* sortie quand T == moi */
    /* la ressource est occupée */
L2: T= ~moi; /* ou 3-moi, T == l'autre */
    /* la ressource est libérée (et est donnée à l'autre) */
```

a) b) d) sont satisfaites

c) ne l'est pas

A et B doivent impérativement occuper alternativement la ressource

Solution partielle 3

- **Les processus partagent un tableau nommé R[2] :**
  - **R[i] == 0 signifie que le processus i n'occupe pas et ne veut pas occuper la ressource**
  - **R[i] == 1 signifie que le processus i occupe ou veut occuper la ressource**

- **Programme 3 :**

```
L1: while (R[~moi] == 1) {;} /* sortie quand R[~moi] == 0 */
L2: R[moi]= 1;
    /* la ressource est occupée */
L3: R[moi]= 0;
    /* la ressource est libérée */
```

b) c) d) sont satisfaites

a) ne l'est pas

Contre-exemple :

A:L1, B:L1, A:L2, B:L2

- **Variante de la solution 3 :**
  - **permutation du test et de l'occupation**

- **Programme 3 bis :**

```
L0: R[moi]= 1;
L1: while (R[~moi] == 1)
    {
L2: R[moi]= 0; /* pour laisser passer l'autre */
L3: R[moi]= 1;
    }
    /* la ressource est occupée */
L4: R[moi]= 0;
    /* la ressource est libérée */
```

a) c) d) sont satisfaites

b) ne l'est pas

Contre-exemple :

Si les tests ne sont jamais fait après L2 et avant L3, alors boucles infinies

Solution complète 4

- **Les processus partagent un tableau nommé R[2] et une variable nommée T :**
  - **R[i] == 0 signifie que le processus i n'occupe pas et ne veut pas occuper la ressource**
  - **R[i] == 1 signifie que le processus i occupe ou veut occuper la ressource**
  - **T == i signifie que le processus i est privilégié pour occuper la ressource**
  - **T != i signifie que le processus i n'est pas privilégié**

- **Programme 4 :**

```
L0: R[moi]= 1;
L1: while (R[~moi] == 1)
    {
L2: if (T == moi) continue;
L3: R[moi]= 0;
L4: while ( T != moi) {;}
L5: R[moi]= 1;
    }
    /* la ressource est occupée */
L6: T= ~moi; /* privilégier l'autre */
L7: R[moi]= 0;
    /* la ressource est libérée */
```

a) car L0 ou L5 et L1

b) car L2 choisit qui passe, L3 laisse passer l'autre, et L6 fait varier le choix

c) car L7 et L1

d) le code est symétrique

Atomicité logiciel

- **Sémaphores (Dijkstra)**
  - **M un entier positif ou nul**
  - **deux fonctions P et V**
  - **soient nP et nV le nombre d'appels faits à P et V**
    - » **1) P et V sont sérialisables**
    - » **2) P ne fait "return" que si : nP - nV ≤ M**
    - » **3) V débloque un processus bloqué dans P (s'il y en a)**
- **non spécifié :**
  - **quand la fonction V débloque un processus**
  - **le choix du processus est laissé à l'implémentation**
- **interprétation :**
  - **M est le nombre initial de ressources**
  - **nP le nombre de ressources consommées**
  - **nV le nombre de ressources produites**

- **Une réalisation :**

```
static int cpt = M;
  /* cpt = M - nP + nV */
static ensemble attente = VIDE;
P()
  {debut_atomique();
  cpt = cpt - 1;
  if (cpt LT 0)
    {bloquer_return();
    ajouter(attente, processus_courant());
    }
  fin_atomique();
  }
V()
  {debut_atomique();
  cpt = cpt + 1;
  if (cpt LE 0)
    {debloquer_return(extraire(attente));
    }
  fin_atomique();
  }
```

- **Propriétés :**
  - **1) cpt peut devenir négatif**
  - **2) cpt = M - nP + nV**
  - **3) soit nS le nombre de processus sortis de P**
    - » **$nS \leq nP$**

- **Théorème :**
  - **$nS = \min(nP , M + nV)$**
- **Démonstration :**
  - **1) à l'initialisation**
    $nS = 0$; $nP = 0$; $nV = 0$; $cpt = M \geq 0$;
  - **2) par récurrence**
    exécution de P ou de V

min avant	nP = nP + 1	effet sur nS	opérande de min
nS == nP et nP < M + nV	nP < M + nV	nS = nS + 1	nS == nP et nP < M + nV
nS = M + nV et nP > M + nV	nP > M + nV	nS = nS	nS == M +nV et nP > M + nV

min avant	nV = nV + 1	effet sur nS	opérande de min
nS == nP et nP < M + nV	nP < M + nV	nS = nS	nS == nP et nP < M + nV
nS = M + nV et nP > M + nV	nP > M + nV	nS = nS + 1	nS == M + nV et nP > M + nV

Famine

- **Définition :**
  - **un processus sera dit en famine de cpu quand il n'évolue plus dans son code**

- **Exemples :**
  - **dans la fonction V la gestion de l'ensemble des processus bloqués est non spécifiée**
    ==>
    avec une gestion Dernier Arrivé Premier Servi,
    - » **le premier mis en attente peut ne jamais être débloqué**

  - **avec HPF (Highest Priority First) et une gestion avec estampille**
    le processus le moins prioritaire peut boucler indéfiniment

Inter-blocage

- **Exemple :**
  - deux personnes Guy et Daniel
  - un comité d'entreprise disposant de forets et d'une perceuse
- **Lundi :**
  - Guy emprunte les forets
  - Daniel emprunte la perceuse
- **Mardi :**
  - Guy demande la perceuse
  - Daniel demande les forets
- **Question :**
  - quel jour Guy et Daniel pourront-ils faire leur trou ?

- **Problème avec des sémaphores :**
  - deux processus A et B
  - 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 bloqué dans P */
    - B : P(S1) /* B est bloqué dans P */
- **Remarque :**
  - c'est un problème général :
    - A est bloqué et c'est B qui peut faire évoluer la situation
    - B est bloqué et c'est A qui peut faire évoluer la situation

  ==>
    - la situation ne peut plus évoluer

- **Remède :**
  - **annuler l'un des appels à P**
    ==>
  - **il faut faire remonter le processus dans son code**
    ==>
  - **il faut pouvoir restaurer les données du processus**
    et
  - **annuler toutes les opérations qu'il a faites**
    et
  - **...**
    ==>
- **C'est le même problème qu'avec les estampilles !**

- **Prévention :**
  - **problème compliqué**
    mais
  - **dans des cas particuliers de gestion de ressources,**
  - **il y a des solutions simples**

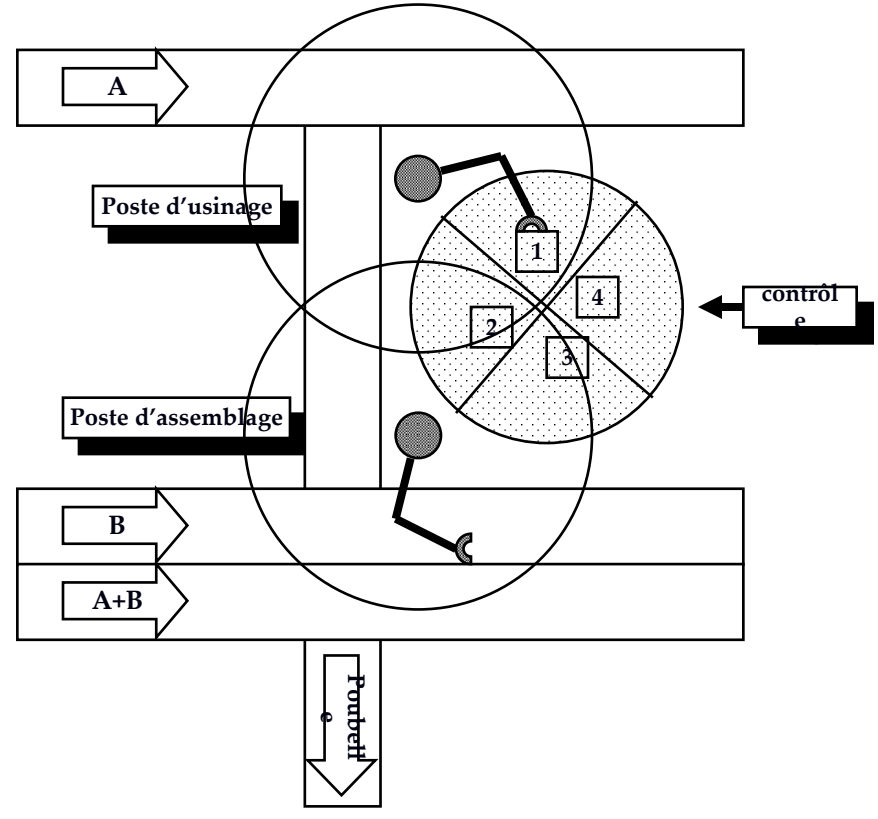
- **Annonce des besoins :**
  - **le processus demande en une seule fois**
  - **toutes les ressources dont il a besoin**

- **Ressources ordonnées partiellement :**
  - **le processus peut faire des demandes successives pourvu :**
    - » **qu'elles portent sur des ressources comparables**
    - » **qu'elles soient faites en suivant la relation d'ordre**
- **Démonstration :**
  - **le processus qui possède une ressource occupée de plus grand rang ne peut pas être bloqué**
- **La condition n'est pas nécessaire :**
  - **A : P(R1) , P(R2) , P(R3)**
  - **B : P(R1) , P(R3) , P(R2)**

- **Sémaphore privé :**
  - **quand chaque processus n'est autorisé qu'à faire une seule des primitives P ou V**
    - » **le sémaphore est dit privé (en fait c'est un emploi particulier)**
- **Interprétation :**
  - **le(s) processus qui fait P**
    - » **attend un signal du(s) 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

- **Un atelier d'assemblage comprend les éléments suivants :**
  - **2 tapis roulants « A » et « B » commandés au pas à pas**
  - **2 tapis roulants « A+B » et « Poubelle » avançant en continu**
  - **1 poste d'usinage pouvant travailler 5 pièces simultanément**
  - **1 poste d'assemblage d'une pièce A avec une pièce B**
  - **1 plateau tournant à 4 bacs numérotés de 1 à 4 et d'un repère**
  - **2 robots RA et RB dont les zones d'interventions sont les disques fins**



- **En utilisant des sémaphores pour la synchronisation, écrire les fonctions *déposer()* pour RA et *prendre()* pour RB avec le contexte suivant :**
  - **RA n'appelle *déposer()* que si une case est libre**
    - » **on dispose de la fonction *poser_la_pièce()***
  - **RB n'appelle *prendre()* que si une case est occupée**
    - » **on dispose de la fonction *enlever_la_pièce()***
  - **Pour RA et RB**
    - » **on dispose de la fonction *examiner()* qui rend à l'appelant s'il doit faire tourner le plateau et prépare l'appel à la fonction *tourner()***
    - » **on dispose de la fonction *tourner()* à appeler après l'appel de la fonction *examiner()***

Problème :

- écrire les fonctions *déposer()* et *prendre()* de sorte que le degré de parallélisme dans les différentes actions soit maximal

- apporter les démonstrations que la conception est correcte