RT5-GL4-IMI4

Systèmes Temps-Réel

Chapitre 2 :

Fondements de la programmation concurrentielle

Olfa Mosbahi

olfamosbahi@gmail.com

Plan du cours

- Programmation concurrente
  - Notion de tache
- Concurrence en Java
- Application
- Conclusion

Programmation concurrente

Motivation:

- le système interagit avec des parties concurrentes de son environnement (le monde réel)
- reconnaître le parallélisme *dans le programme* mène à une structure plus naturelle
  - dans le cas contraire, le programme est rendu séquentiel de façon plus ou moins arbitraire
  - …par exemple en exécutant une boucle:

```
TQ vrai FAIRE
    si evt1 : exécute tâche 1
    si evt2 : exécute tâche 2
    …
FIN TQ
```

⇒ la structure n'a rien à voir avec le pb. initial

⇒ la décomposition du problème est plus difficile

⇒ la parallelisation sur plus d'un processeur est difficile

Notion de tâche

- le *système* est une collection de tâches autonomes exécutant en parallèle (parallélisme logique)
- une tâche
  - (statiquement) est décrite par un programme séquentiel
  - (à l'exécution) a son propre fil de contrôle
  - peut *communiquer* et partager des *ressources* avec d'autres tâches
- l'interprétation varie au cours du cycle de développement:
  - analyse / conception : tâche = une série d'actions déclanchée par un stimulus venant d'un capteur **ou** par l'arrivée d'une échéance temporelle
  - implémentation : tâche = processus ou thread ou co-routine ou objet actif ou …

Processus et threads

Processus → fil de contrôle exécuté dans sa propre machine virtuelle (espace mémoire, handle de fichiers, etc.)

- tous les OS fournissent une notion de processus

Thread → fil de contrôle exécuté en parallèle d'autres threads dans un même processus

⇒ partage de la mémoire, des handles de ressource, …

- construction fournie dans certains OS

Threads et langages de programmation:

- certains langages fournissent aussi une notion de thread
- débat sur la question si les langages doivent fournir des constructions pour la concurrence
  (Ada, Java : oui ; C, C++ : non)

Constructions de programmation concurrente

Permettent d'exprimer:

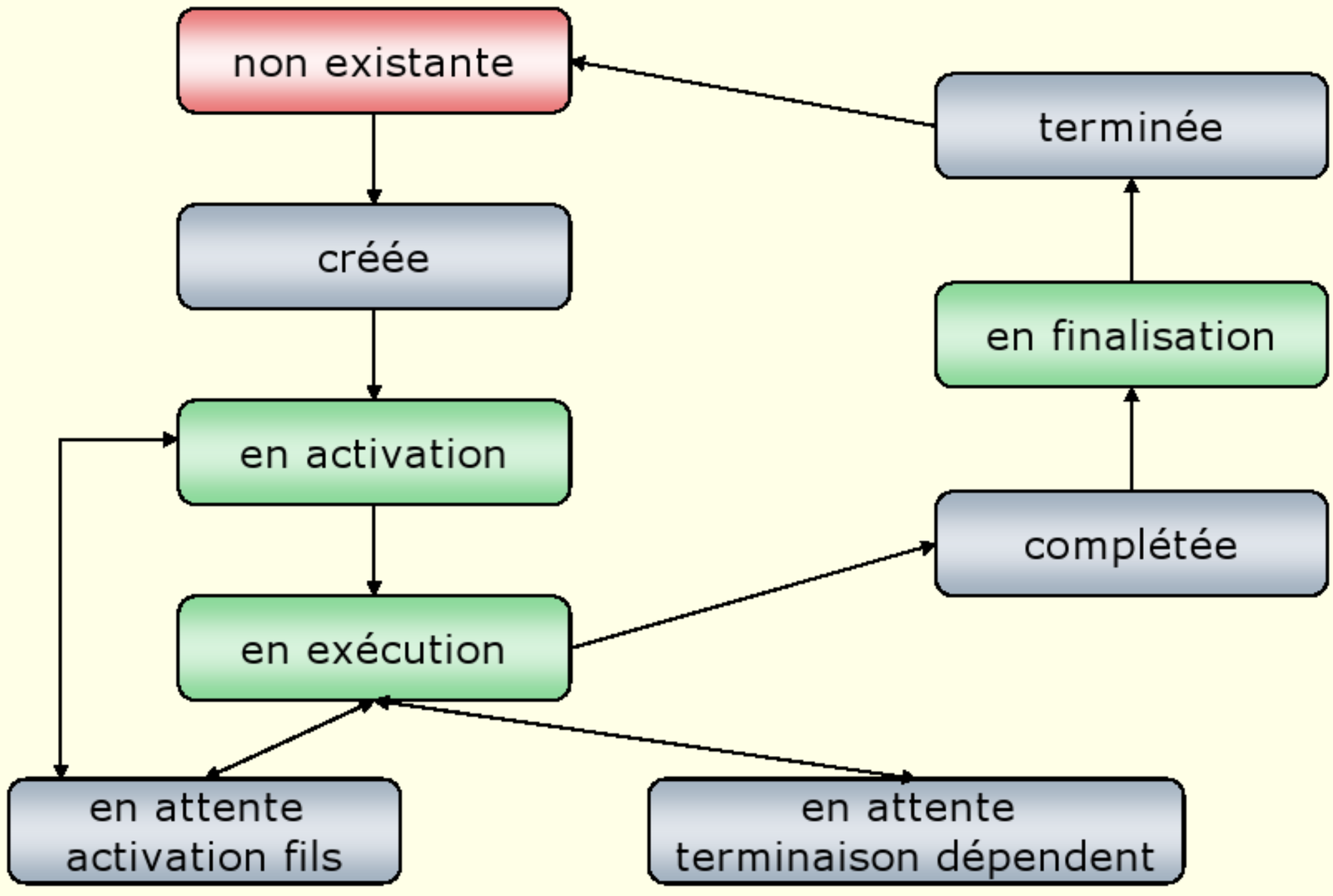
- l'existence ou la création de tâches concurrentes
- la communication entre tâches
- la synchronisation des tâches

Le tâches peuvent être:

- indépendantes
- coopérantes (⇒ besoin de communiquer)
- en compétition (⇒ besoin de se synchroniser)

Une Tâche:
• peut créer des Tâches « Fils », elle se termine après leurs terminaisons
• peut dépendre de la fin d'exécution d'autres tâches

Diagramme d'Etat d'une Tâche



Hiérarchie de tâches

- Parent et maître d'une tâche
  - Parent = celui qui la crée
  - Le maître est en général le parent, mais pas toujours
- A la création d'un fils, le parent est suspendu jusqu'à son activation
- Le maître attend que ses dépendants se terminent avant de se terminer
- Une tâche peut tester si une autre est terminée

Plus sur la concurrence en Java

- classe prédéfinie *java.lang.Thread*
  - créer un nouveau thread: `t = new Thread(r)`
  - démarrer son exécution : `t.start()`
- qui est **r** ?
  - interface prédéfinie :

```
public interface Runnable {
  public abstract void run();
}
```

  - **r** est une instance de classe qui implémente *Runnable*

Thread

```
public class Thread extends Object implements Runnable
{
    public Thread();
    public Thread(Runnable target);
    public void start() ;
    public void run() ;
    public static void sleep(long millis) ;
    public static Thread currentThread() ;
    public void destroy() ;  // not implemented
    public void interrupt() ;
    public final void join(long millis) ;
    public final void stop() ;// deprecated
    public final native boolean isAlive () ; // deprecated
    …
}
```

hiérarchies et groupes sont supportés, mais pas de notion de master

Création de tâches

- statique ou dynamique
  - statique : toutes les tâches sont créées dès le démarrage du système.
    Elles ne changent pas durant toute l'exécution.
  - dynamique : on peut créer des tâches en cours d'exécution.

Création de tâches

- …ou explicite

exemple en Java:

```
// déclaration de tâche
public class A extends
        java.lang.Thread
{
   public void run()
   {
       // prog séquentiel pour A
   }
}


// création explicite
A a = new A();
a.start();
```

Le modèle fork / join

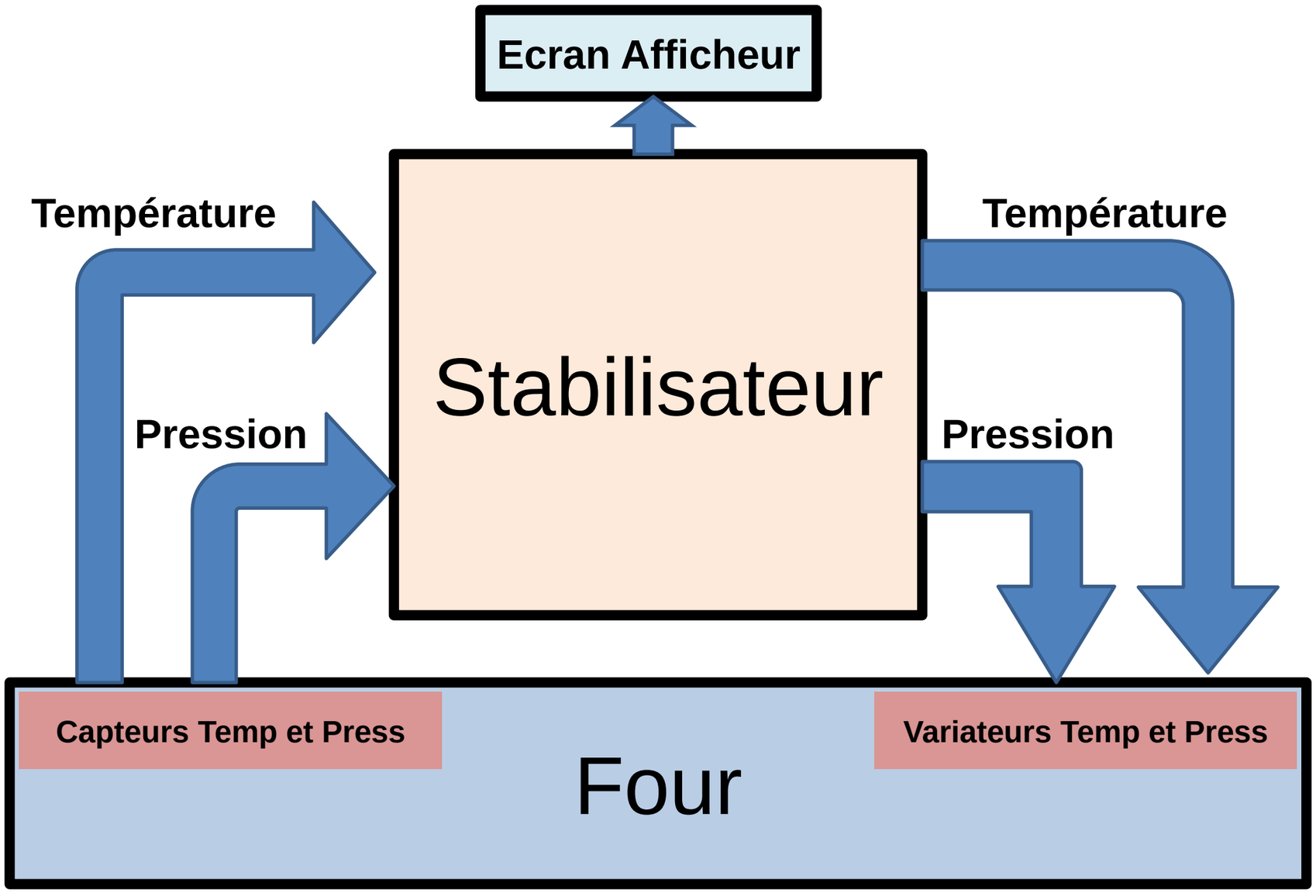
- ***fork*** spécifie qu'une tâche commence à s'exécuter en parallèle avec l'appelant
- ***join*** permet à l'appelant d'attendre (en état *suspendu*) la fin de la tâche crée
- variantes
  - Java :

```
A a = new A();
a.start();        // ceci est un fork
...
a.join();         // attendre la fin de a
```

  - Unix / Posix :

```
pid = fork();
if( NULL == pid ) {
  ...      // on est dans le fils
} else {
  ...      // on est dans le parent
}
```

Application: Stabilisateur de Température et Pression dans un Four d'une Cimenterie



Architectures possibles

- un programme séquentiel unique
  → on ignore la concurrence logique de T, P
  → aucun support OS n'est nécessaire

- T, P écrites dans un langage séquentiel
  (comme procedures ou comme programmes distincts)
  → primitives de l'OS pour créer les tâches et communiquer

- Un prog. concurrent qui préserve la structure logique de T, P . Un OS ou autre run-time est nécessaire.

Solution séquentielle

```
class controleur {
  CT : Capteur_Temp;
  CP : Capteur_Pres;
  OT : Commande_Temp;
  OP : Commande_Pres;
  ...
  int main(String[] args) {
    while(true) {                    // boucle infinie
          CT.read();                                  // lecture de l'ADC (bloquant)
          ConversionTemp(CT, OT);                     // calcul
          OT.write();                                 // envoi au brûleur
          CT.writeToScreen();                         // affichage

          CP.read();                                  // lecture de l'ADC (bloquant)
          ConversionPression(CP, OP);                 // calcul
          OP.write();                                 // envoi à la valve
          CP.writeToScreen();                         // affichage
    }
  }
}
```

Discussion

- la lecture de température et de pression se fait au même rythme
- on peut y remédier avec des compteurs
- mais dans ce cas, on a probablement besoin de fragmenter *ConversionTemp* et *ConversionPression* pour effectuer le travail
  - exemple :
    - *ConversionTemp* doit s'exécuter toutes les 1ms et prend 0.5ms
    - *ConversionPression* doit s'exécuter toutes les 2ms et prend 1ms
    - ⇒ c'est faisable, mais il faut sectionner *ConversionPression* en 2 tranches (égales!)
- pendant qu'on fait *CT.read()*, on ne peut pas s'occuper de la pression

⇒ *si le capteur température tombe en panne, on ne regarde plus le capteur pression non plus!!*

Solution séquentielle améliorée

```
class controleur {
  CT : Capteur_Temp;
  CP : Capteur_Pres;
  OT : Commande_Temp;
  OP : Commande_Pres;
  TempReady : boolean;
  PresReady : boolean;
  ...
  int main(String[] args) {
    while(true) {                    // boucle infinie
      if(TempReady) {                // polling -- attente active
        CT.read();
        ConversionTemp(CT, OT);
        OT.write();
        CT.writeToScreen();
      }
      if(PresReady) {
        CP.read();
        ConversionPression(CP, OP);
        OP.write();
        CP.writeToScreen();
      }
  }
```

Discussion

- solution plus fiable
- …mais terriblement inefficace – ça occupe le processeur tout le temps pour regarder 2 booleens
- la fragmentation du calcul reste nécessaire si les fréquences de réponse l'impose
- difficile à généraliser pour des systèmes plus conséquents

Solution utilisant les tâches OS

processus T:

```
…
void main() {
 while(true) {
   CT.read();
   ConversionTemp(CT, OT);
   OT.write();
   CT.writeToScreen();
 }
}
```

processus P:

```
…
void main() {
 while(true) {
   CP.read();
   ConversionPres(CP, OP);
   OP.write();
   CP.writeToScreen();
 }
}
```

processus root:

```
…
void main() {
  if( fork("P") ) if( fork ("T") ) return 0;
}
```

Discussion

- c'est mieux :
  - on a déjà les tâches bien identifiées
  - on peut les annoter avec des fréquences, etc.
  - on peut instruire l'OS pour les ordonnancer selon les besoins
- en réalité, il faut probablement encore un processus pour partager la ressource « écran » correctement
- utilisation des primitives de communication entre processus
- le programme devient vite difficile à maintenir!!

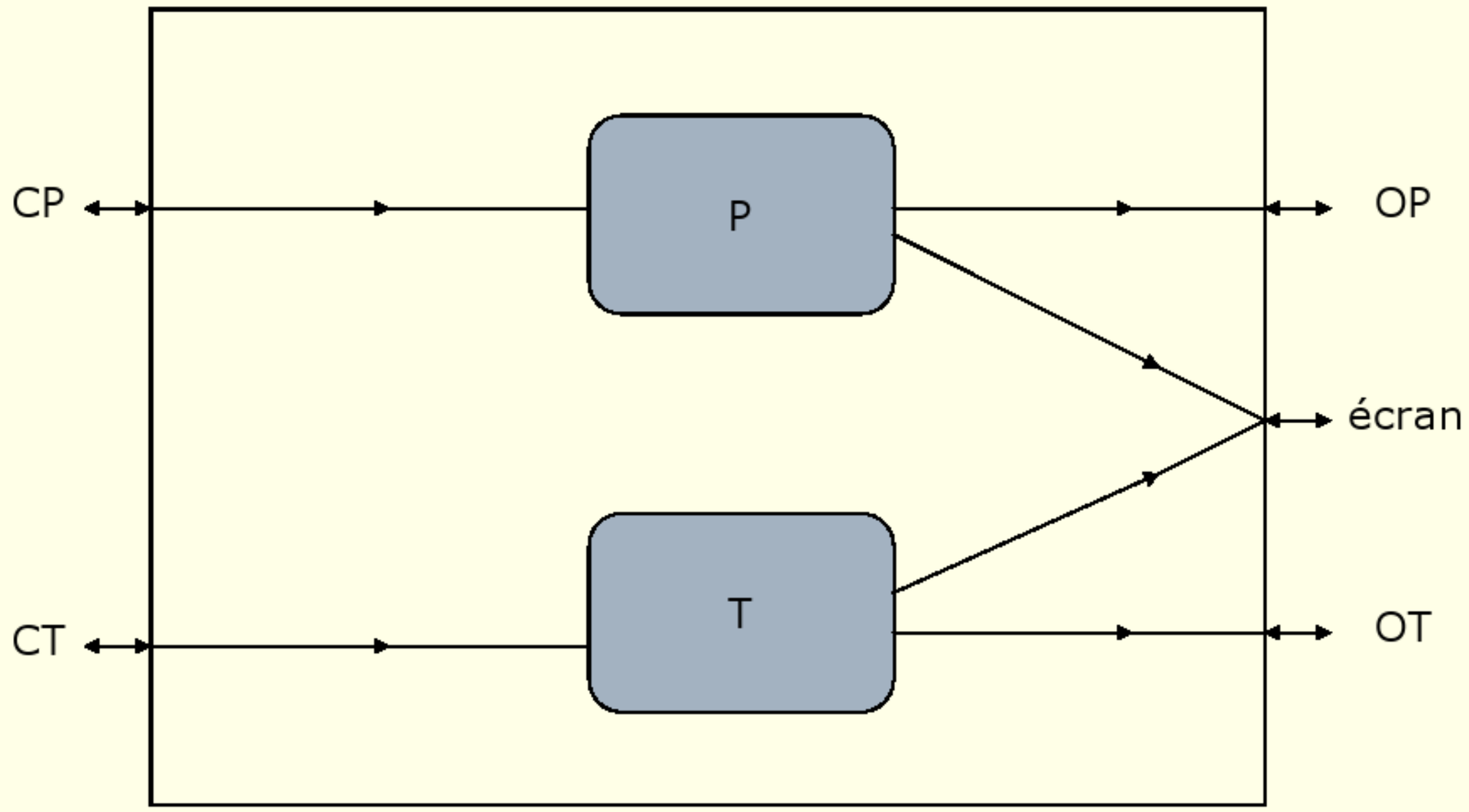
Solution dans un langage concurrent (Java)

```
class T implements Runnable {
void run() {
 while(true) {
   CT.read();
   ConversionTemp(CT, OT);
   OT.write();
   CT.writeToScreen();
 }
}
```

```
class P implements Runnable {
void run() {
 while(true) {
   CP.read();
   ConversionPres(CP, OP);
   OP.write();
   CP.writeToScreen();
 }
}
```

```
class root {
int main(String[] args) {
 Thread t1 = new Thread( new T() );
 Thread t1 = new Thread( new P() );
}
```

Solution dans un langage concurrent (SDL)



Discussion

- les tâches sont identifiées, la logique de l'application est apparente dans la structure du programme
- la communication et la synchronisation sont souvent plus faciles à exprimer qu'en faisant des appels OS
- quand une tâche est suspendue dans un *read*, l'autre peut exécuter
- si les deux sont suspendues, l'attente est *passive* (ne consomme pas de CPU)

concurrence dans l'OS vs. concurrence dans les langages

- pour la concurrence dans les langages
  - programmes plus lisibles / faciles à maintenir
  - indépendance de l'OS, les programmes sont plus portables
  - un ordinateur embarqué n'a peut-être pas de OS !
- contre
  - plus facile à composer des programmes de langages différents s'ils utilisent les mêmes principes (ceux de l'OS)
  - l'implémentation d'un langage concurrent peut ne pas être efficace sur tous les OS
  - les standards de OS existent (Posix)

Conclusion sur la prog. concurrente

- les domaines d'application des systèmes temps réel sont généralement parallèles
- la concurrence disponible directement dans le langage de programmation rend la vie plus facile
- sans concurrence, un système TR est en général une boucle infinie
  - la structure de la boucle ne montre pas la structure logique des tâches
  - on ne peut pas préciser les caractéristiques des tâches (période, temps de réponse) sans les identifier
  - la solution de passe pas à l'échelle de grandes applications