Par Iulian Ober, IRIT

RT5-GL4-IMI4

Systèmes Temps-Réel

Chapitre 3 :

Synchronisation et Communication des tâches

Olfa Mosbahi

olfamosbahi@gmail.com

Plan du cours

- Synchronisation
  - Attente active
  - Sémaphores
- Moniteurs
- Communication
- Conclusion

Synchronisation et communication

- le comportement d'un prog. concurrent dépend de la synchronisation et de la communication entre ses tâches

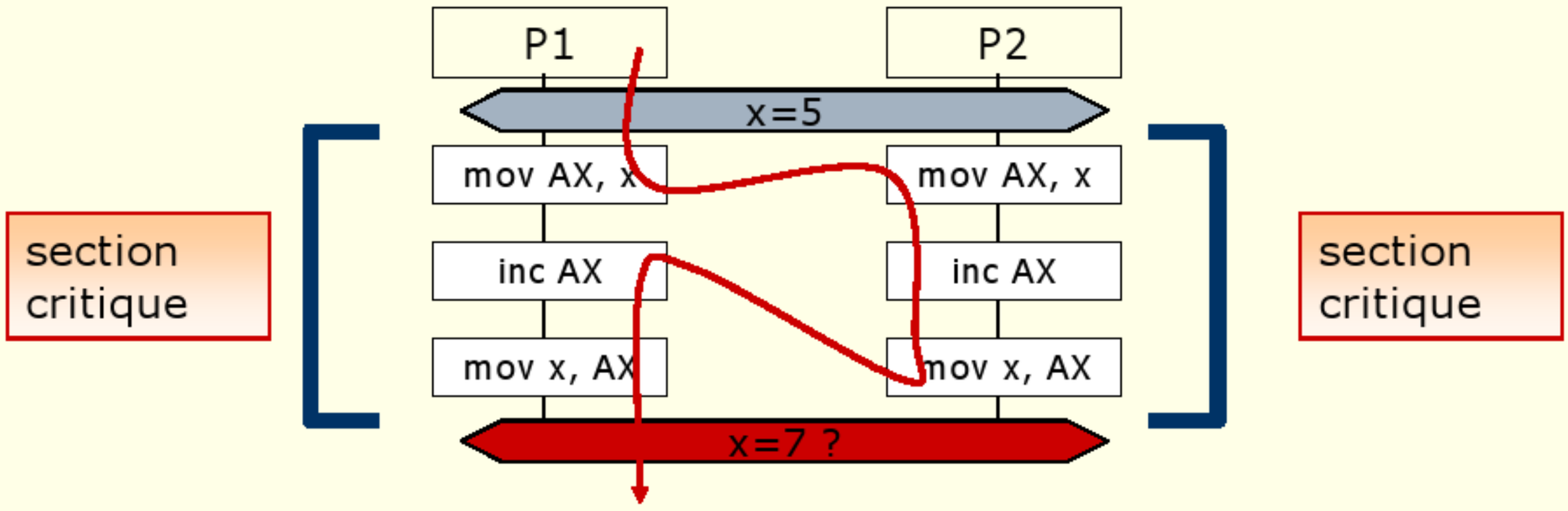
Synchronisation = satisfaction de contraintes sur l'entrelacement des actions de plusieurs tâches (e.g., action A de la tâche P s'exécute uniquement après action B de R)

Communication = passage d'informations d'une tâche à une autre

- concepts liés
- la communication est basée sur une mémoire commune ou sur le passage de messages

Éviter l'interférence

- l'exécution indivisible des séquences d'instructions qui accèdent à des variables partagées est nécessaire:

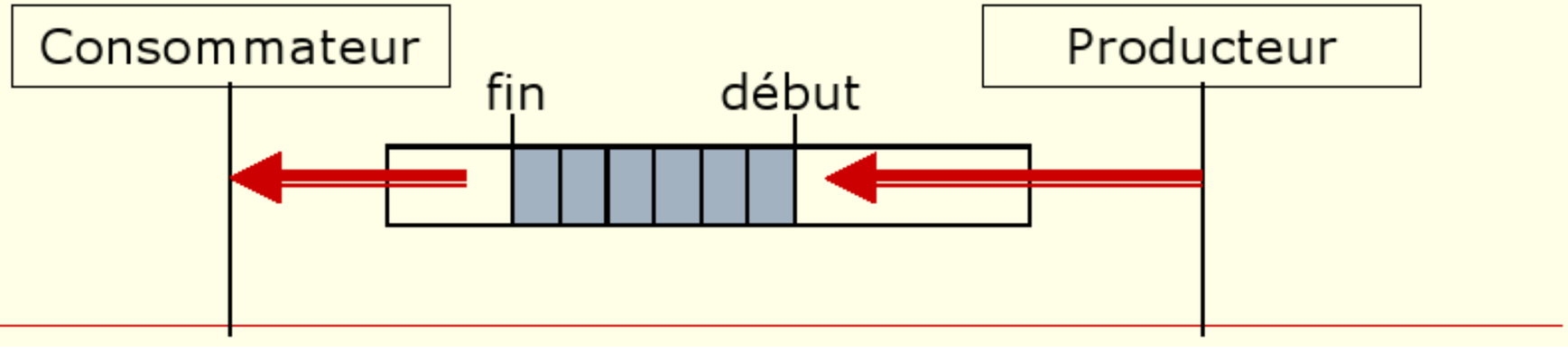


- hypothèse : « mov x, AX » est indivisible !
  (hypothèse pas valable pour les données structurées)

2 types de synchronisation

- Exclusion mutuelle
  - assurance pour une tâche que l'exécution d'une séquence d'instructions se fait sans interférence
  - la séquence d'instructions s'appelle section critique

- Attente conditionnelle
  - une tâche doit exécuter une action uniquement après une autre action d'une autre tâche
  - exemple : un tampon fini ⇒ deux attentes cond.



Exigences pour les sections critiques

1. Exclusion mutuelle
2. Compétition constructive
   - les tâches ne **s'empêchent** pas réciproquement d'entrer
   - les tâches ne s'invitent pas réciproquement à entrer à l'infini
3. Indépendance
   - si une tâche est plus lente (dans sa partie non critique) cela n'a pas d'effet sur l'entrée des autres tâches dans **leurs** sections critiques
4. Équité
   - une tâche ne peut être retardée à l'infini à l'entrée de sa section critique

Synchro avec attente active

- utiliser une variable partagée comme un drapeau
  - Consommateur :

```
while (tamponVide) {}
//…consommation
if(fin>debut) tamponVide=true ;
```

attente active

  - Producteur :

```
//…production
tamponVide = false ;
```

- fonctionne pour l'attente conditionnelle, mais pas facile pour l'exclusion mutuelle

Exclusion mutuelle avec attente active

fausse solution no. 1

T1:
```
while(true) {
    flag1 = true;
    while(flag2){} // attente active
    …
    // section critique
    …
    flag1 = false;
}
```

T2:
```
while(true) {
    flag2 = true;
    while(flag1){} // attente active
    …
    // section critique
    …
    flag2 = false;
}
```

⇒ possibilité d'interblocage !

Exclusion mutuelle avec attente active

fausse solution no. 2

T1:
```
while(true) {
    while(flag2){} // attente
    flag1 = true;
    …
    // section critique
    …
    flag1 = false;
}
```
active

T2:
```
while(true) {
    while(flag1){} // attente
    flag2 = true;
    …
    // section critique
    …
    flag2 = false;
}
```
active

⇒ pas d'exclusion mutuelle !

Exclusion mutuelle avec attente active

fausse solution no. 3

T1:

```
while(true) {
    while(tour == 2){}
    …
    // section critique
    …
    tour = 2;
    Sec. Non-critique…
}
```

Attente active

T2:

```
while(true) {
    while(tour == 1){}
    …
    // section critique
    …
    tour = 1;
    Sec. Non-critique…
}
```

Attente active

⇒ les tâches doivent revenir à la section critique au même rythme

Solution : Algorithme de Peterson

L'**algorithme de Peterson** est un algorithme d'exclusion mutuelle pour la programmation concurrente. Il est basé sur une approche par attente active. Il est constitué de deux parties :

- Le protocole d'entrée dans la section critique et
- Le protocole de sortie.

L'algorithme présenté est une version pouvant fonctionner avec deux threads. Il a été publié par Gary Peterson en 1981.

Algo de Peterson (1981)

T1:

```
while(true) {
   flag1 = true;
   tour = 2;
   while(tour==2  && flag2);
   …
   // section critique
   …
   flag1 = false;
}
```

Attente active

T2:

```
while(true) {
   flag2 = true;
   tour = 1;
   while(tour==1  && flag1);
   …
   // section critique
   …
   flag2 = false;
}
```

Attente active

Éviter l'attente active: sémaphores

- mécanisme de synchronisation composé de
  - une variable entière *n*
  - une file d'attente *f*
    - si *n* > 0 alors n est le nombre de tâches qui peuvent passer par le sémaphore avant qu'il devienne rouge
    - si n = 0, alors les tâches qui essayent de passer se rangent dans la file *f* et deviennent suspendues (attente passive)
  - n et f sont accédées uniquement par 2 opérations:
    - WAIT (ou P) = la tâche courante essaye de passer

      si n> 0 alors n=n–1 et retour immédiat
      sinon suspension dans la file *f*
    - SIGNAL (ou V) = la tâche courante libère le sémaphore

      si f non vide, choix d'une tâche suspendue pour reprise

      si f vide, n=n+1

n : nombre de taches qui peuvent accéder en même temps à la ressource

Attente conditionnelle avec sémaphore

var consyn : sémaphore (* initalisé à 0 *)

tâche T1:

```
…
instruction X;
wait(consyn);
instruction Y;
…
```

tâche T2:

```
…
instruction A;
signal(consyn);
instruction B;
…
```

T2 puis T1

wait → P

signal → V

Dans quelle ordre s'exécutent les instructions?

Exclusion mutuelle avec sémaphore

var mutex : sémaphore (* initalisé à 1 *)

tâche T1:

```
…
instruction X;
wait(mutex);
instruction Y;
signal(mutex);
instruction Z;
…
```

tâche T2:

```
…
instruction A;
wait(mutex);
instruction B;
signal(mutex);
instruction C;
…
```

La premiere tache qui se presente aura le processeur et bloque l'autre tache

Dans quelle ordre s'exécutent les instructions?

Deadlock

var X,Y : sémaphore (* initalisé à 1 *)

tâche T1:

```
...
wait(X);
wait(Y);
  // instruction protégées
...
```

tâche T2:

```
...
wait(Y);
  wait(X);
  // instruction protégées
...
```

⇒ attente cyclique !

Régions Critiques Conditionnelles (CCR)

- grouper les données partagées dans des ressources
- une CCR est une portion de code en exclusion mutuelle sur une ressource
- … l'entrée dans une CCR peut être gardée par une condition sur la ressource

resource buf : Buffer

tâche Prod:

```
…
region buf when buf.size < N do
…
end region
```

tâche Cons:

```
…
region buf when buf.size > 0 do
…
end region
```

Moniteurs

Objectif: grouper les régions critiques (pour une ressource) dans un seul endroit du code

- les régions critiques
  - → procédures d'un module (le moniteur)
  - → les appels sont sérialisés
- les variables partagées
  - → cachées par le moniteur
  - → accessibles uniquement par les procédures

Exemple

```
monitor buffer;
   export prod, cons;
      ...variables...

      procedure prod (D : Data);
            ...
      end;

      procedure cons (var D : Data);
            ...
      end;
      begin
        ... instructions d'initialisation
end;
```

attente conditionnelle dans le moniteur
- par des sémaphores (!)
- par des variables « condition » (dans moniteurs de Hoare, 1974)

Exemple, avec conditions

```
monitor buffer;
  export prod, cons;
     ...variables...
     nonvide, nonplein : condition;

     procedure prod (D:Data)
        while(n=size) then wait(nonplein);
        ...
        signal(nonvide)
     end;

     procedure cons (var D : Data)
        while(n=0) then wait(nonvide);
        ...
        signal(nonplein);
     end;
     begin
       ... instructions d'initialisation
end;
```

Condition ≠ sémaphore

- **wait** bloque toujours la procedure
- **wait** libère le moniteur

Problème : que fait **signal**?

? termine la procédure appelante

? libère le moniteur

? ne libère pas le moniteur

wait → P

signal → V

Moniteurs en Java

- Java fournit un concept de moniteur dans le contexte des objets ou de classes
- Chaque objet ou classe (les classes sont des objets!) a par défaut un mutex (lock)
  - pas directement accessible
  - bloqué/débloqué par l'appel à des méthodes étiquetées **synchronized**
  - bloqué/débloqué par l'entrée dans un bloc

```
synchronized(obj) {
   …
}
```

⇒ les méthodes synchronisées ont accès exclusif à leur objet

⇒ les méthodes non synchronisées peuvent s'exécuter à tout moment

Exemple

```
public class IntegerBuffer
{
  ...

  public synchronized void write(int i)
  {
    ...
  };

  public synchronized int read()
  {
    ...
  };
}
```

Wait et Notify

- pour la synchronisation conditionnelle on a (dans *java.lang.Object*) :
  - public void wait() throws InterruptedException, IllegalMonitorStateException;
  - public void notify() throws IllegalMonitorStateException;
  - public void notifyAll() throws IllegalMonitorStateException;
- ne peuvent être appelées que si le thread détient le *lock*
- ...sinon ⇒ IllegalMonitorStateException

Wait et Notify

- wait suspend le thread et libère le lock
- notify réveille un thread (quelconque), mais ne libère pas le *lock*. Le thread réveillé doit attendre que le *lock* soit libéré
- notifyAll réveille tous les threads en wait
- un thread peut être réveillé aussi par thread.interrupt()
  (mais le réveil est plus brutal ⇒ InterruptedException)

Synchronisation conditionnelle

- entre notify et la reprise (sortie de wait) une méthode peut avoir été exécutée par un autre thread

⇒ même si cond=true au moment de notify, il se peut que cond=false après la sortie de wait

Conclusion:

a la sortie d'un wait, on doit toujours re-évaluer la condition:

```
while(cond==false) wait();
```

Exemple du tampon fini

```
public class BoundedBuffer  {
  private int buffer[];
  private int first;
  private int last;
  private int numberInBuffer = 0;
  private int size;

  public BoundedBuffer(int length) {
    size = length;
    buffer = new int[size];
    last = 0;
    first = 0;
  };
```

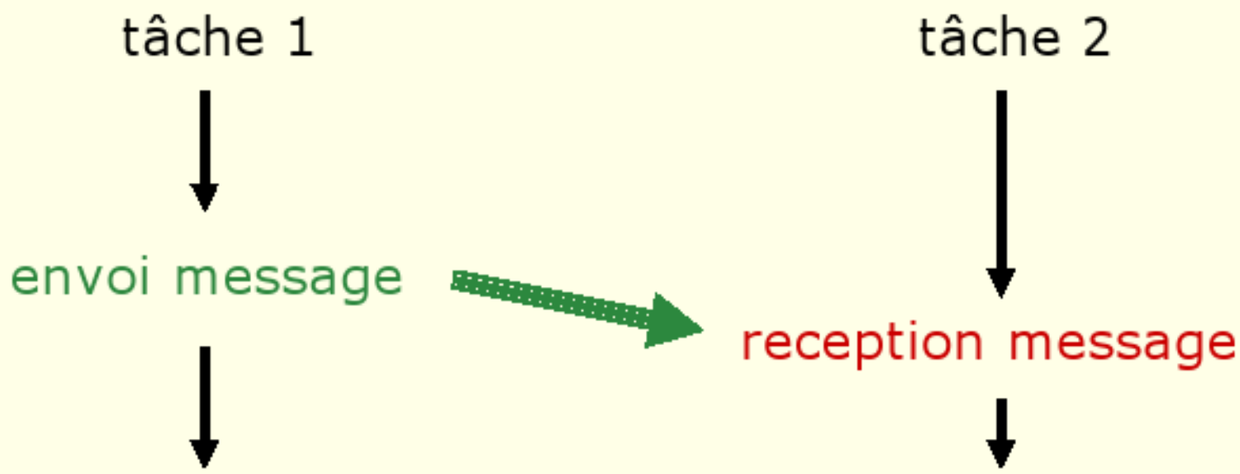
Exemple du tampon fini

```
public synchronized void put(int item)
      throws InterruptedException
{
   while (numberInBuffer == size) wait();
   last = (last + 1) % size ;
   numberInBuffer++;
   buffer[last] = item;
   notify();
};

public synchronized int get() throws InterruptedException
{
   while (numberInBuffer == 0) wait();
   first = (first + 1) % size ; // % is modulus
   numberInBuffer--;
   notify();
   return buffer[first];
};
```

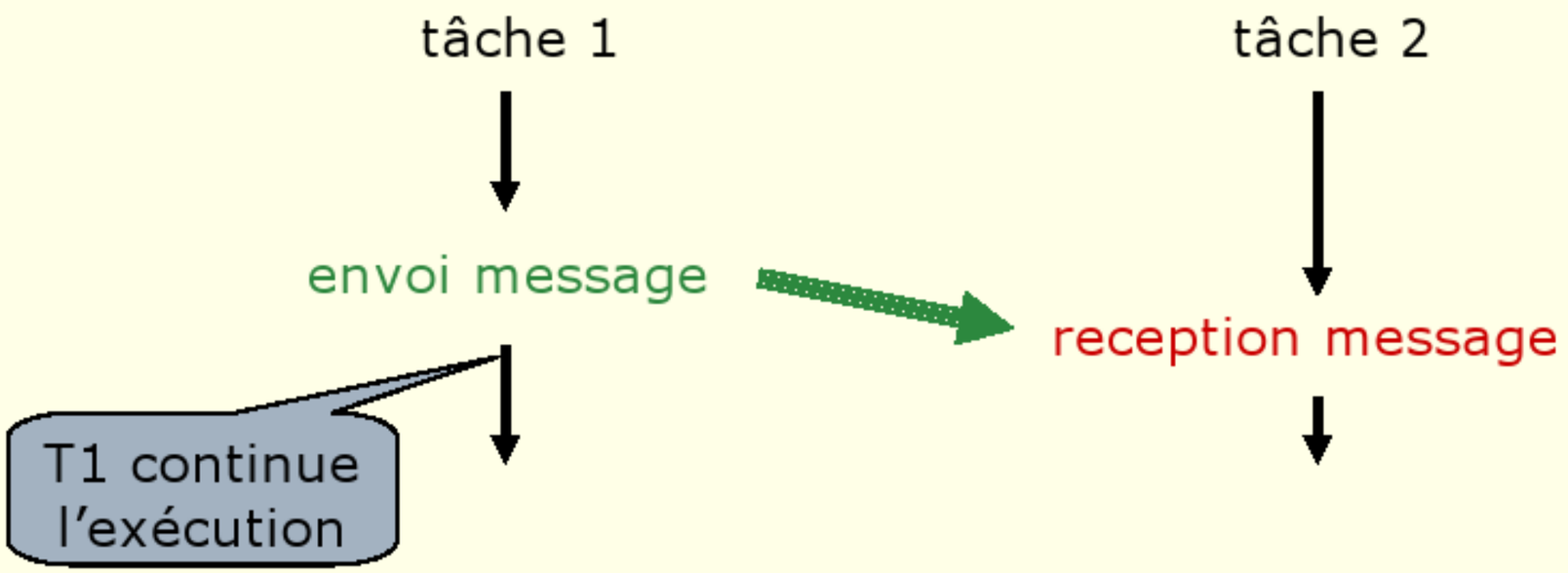
Communication et synchronisation par messages

Communication et synchronisation par messages



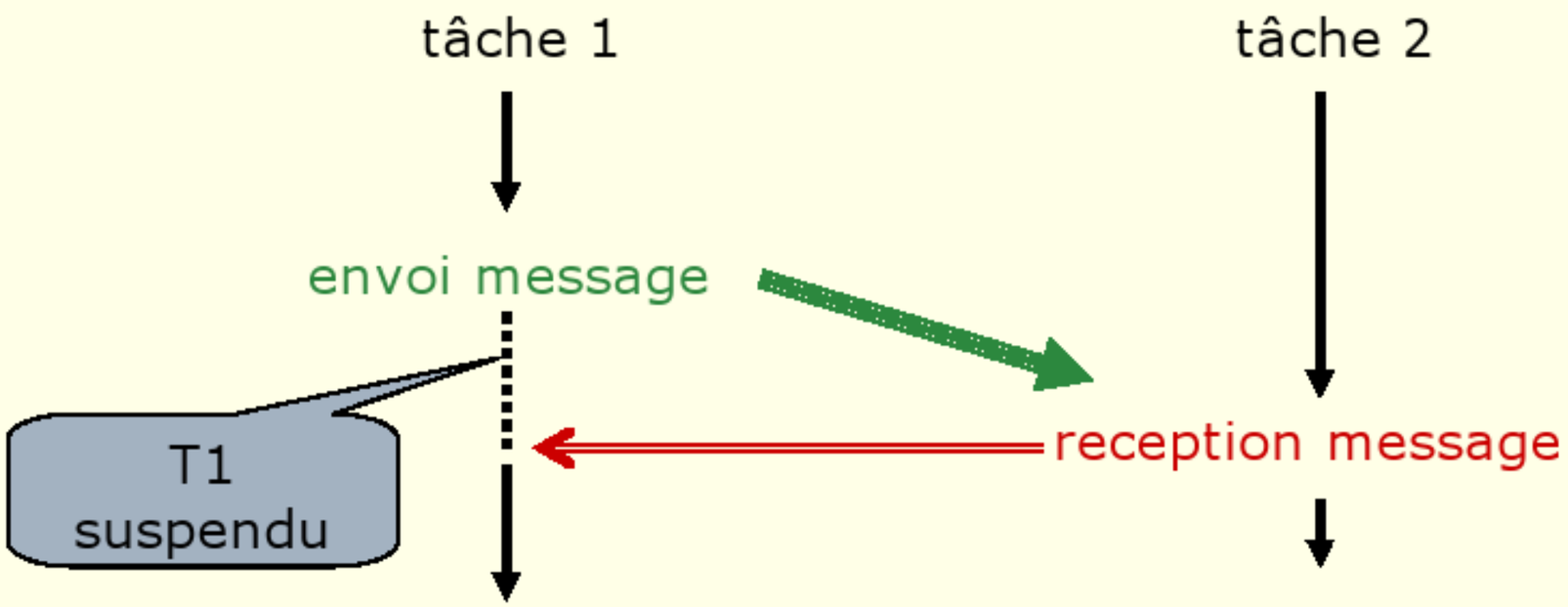
Question principale: le modèle de synchronisation (i.e., qui attend qui?)

Message asynchrone



⇒ nécessite un tampon pour stocker les messages !

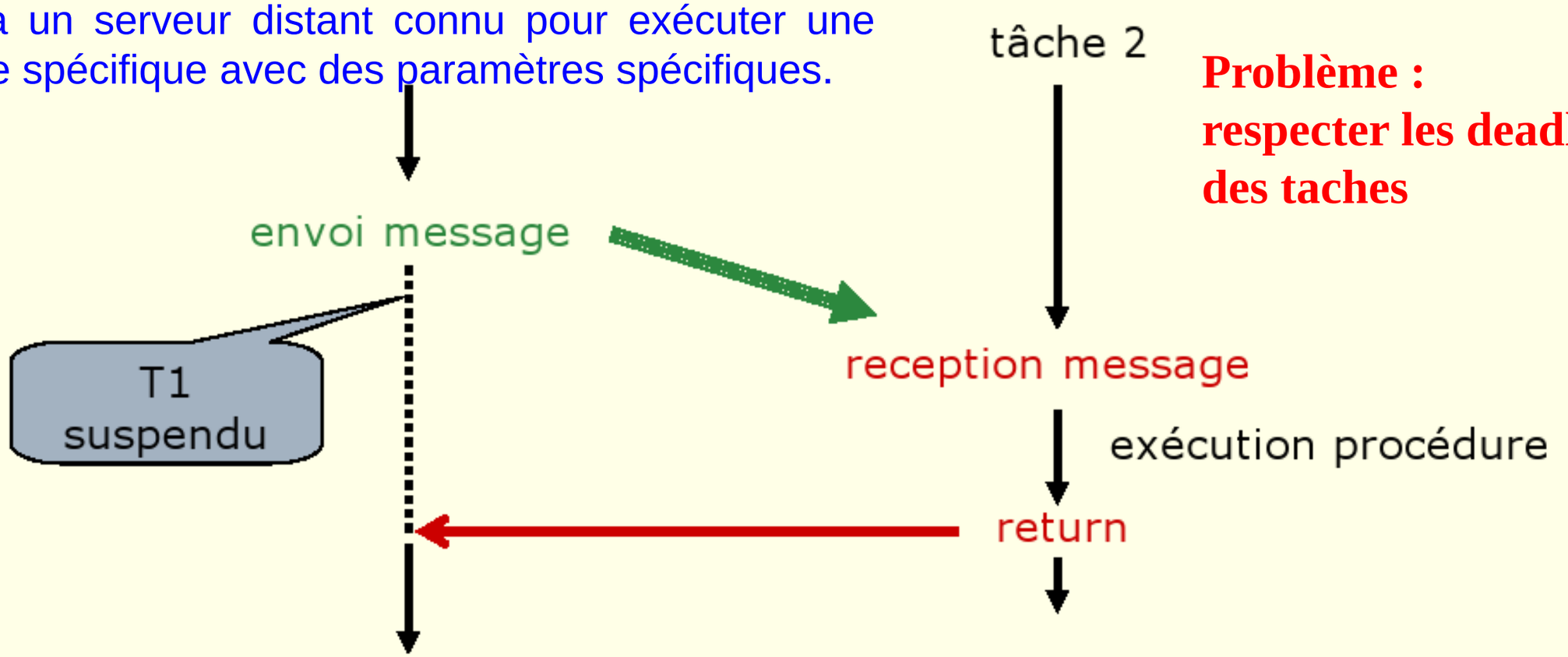
Message synchrone



- pas besoin de tampon
- a.k.a. rendez-vous

Appel de procédure distante (RPC)

Un RPC est initié par le *client* qui envoie un message de requête à un serveur distant connu pour exécuter une procédure spécifique avec des paramètres spécifiques.



Problème :
respecter les deadlines des taches

- a.k.a rendez-vous étendu (Ada)
- peut être implémenté par 2 échanges de messages (synchrone ou asynchrone)

Résume (synchronisation et communication)

- principaux types de synchronisation : exclusion mutuelle et attente conditionnelle
- section critique : partie de code qui doit s'exécuter en exclusion mutuelle
- mécanismes de synchronisation :
  - algos basés sur l'attente active et la mémoire partagée
  - sémaphores
  - régions critiques conditionnelles
  - moniteurs
- attente dans les moniteurs → variables « condition »
- POSIX : sémaphores, moniteurs (mutex), conditions
- Java : moniteurs/conditions dans le contexte objet

Résume (communication par messages)

- Formes:
  - message asynchrone (pas d'attente à l'envoi)
  - message synchrone (attente – synchronisation)
  - appel de procédure distante

- Attente sélective: capacité d'une tâche à attendre plusieurs types de messages en même temps

- Propriété des files de messages

Besoins en facultés temporelles

- Interface avec les notions liées au temps
  - accéder à des horloges pour mesurer le passage du temps
  - retarder un processus jusqu'à un moment ultérieur
  - programmer une temporisation (*time-out*) pour traiter l'absence d'un événement

- Représentation des exigences temporelles
  - spécification des périodes des tâches
  - spécification des échéances (*deadline*)

Types de données et opérations

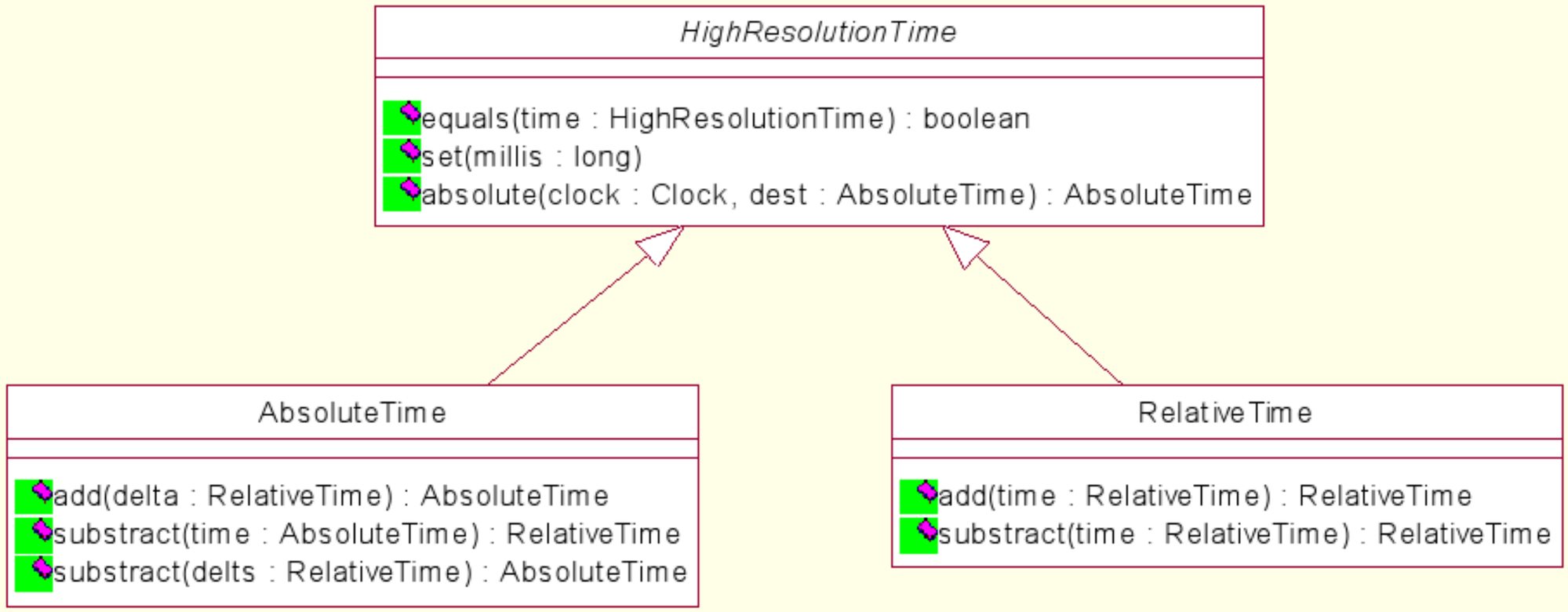
- *« Temps absolu »*
  - identifie de manière unique chaque moment
  - exemple : *Date : Heure*
  - autre exemple : *nombre de millisecondes depuis 1/1/1970 0h GMT (Java)*

- *« Durée relative »*
  - désigne la distance entre deux moments

- exemples d'opérateurs usuels :

  \+ : Temps × Durée ⟶ Temps
  – : Temps × Durée ⟶ Temps
  – : Temps – Temps ⟶ Durée
  * : Integer × Durée ⟶ Durée
  maintenant : ⟶ Temps
  tick : ⟶ Durée
  …

Le temps en RT Java



Le temps en RT Java

```
public abstract class Clock
{
  public Clock();
  public static Clock getRealtimeClock();
  public abstract RelativeTime getResolution();
  public AbsoluteTime getTime();
  public abstract void getTime(AbsoluteTime time);
  public abstract void setResolution(RelativeTime resolution);
}
```

Le temps en POSIX

- type clockid_t
  - constante CLOCK_REALTIME identifie l'horloge temps réel
    (donne le temps relatif à 1/1/1970 0h GMT)
- struct timespec : temps relatif ou absolu
- quelques fonctions:

```
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);
int clock_getres(clockid_t clock_id, struct timespec *res);
int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

Besoins en facultés temporelles

- Interface avec les notions liées au temps
  - accéder à des horloges pour mesurer le passage du temps
  - retarder un processus jusqu'à un moment ultérieur
  - programmer une temporisation (*time-out*) pour traiter l'absence d'un événement

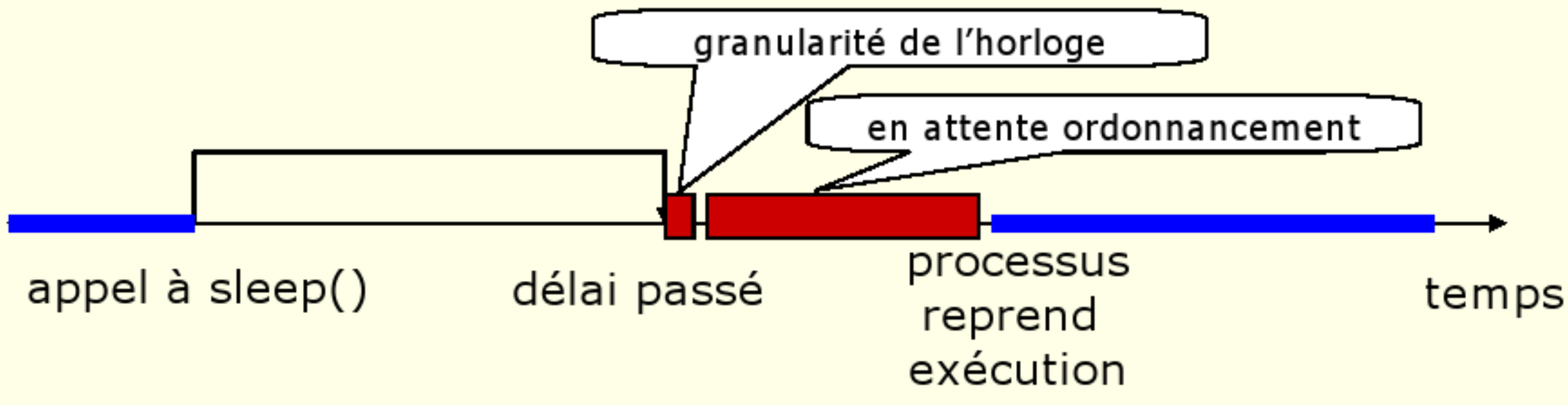
- Représentation des exigences temporelles
  - spécification des périodes des tâches
  - spécification des échéances (*deadline*)

Retarder un processus

- objectif : retarder sans occuper le processeur
- Java :
  - Thread :

    static void sleep(long millis, int nanos)
  - javax.realtime.RealtimeThread :

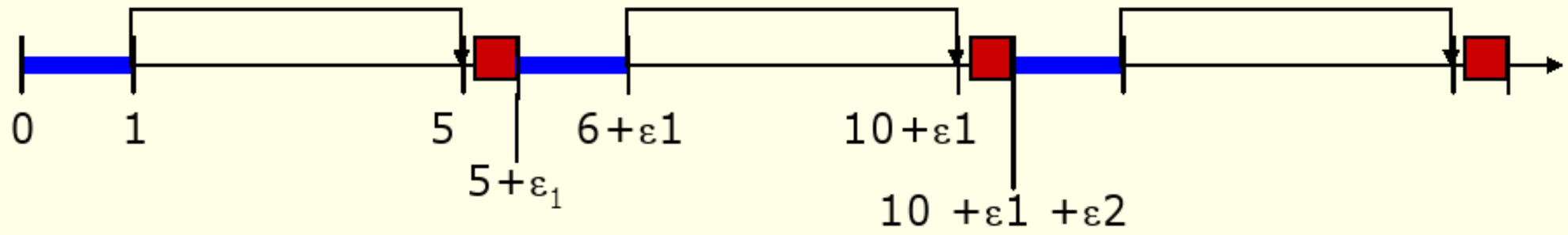
    public static void sleep(Clock clock, HighResolutionTime time)
- *Le délai est un minimum, pas un maximum !*



Accumulation des déviations (*drift*)

- exemple : exécuter une action toutes les 5ms

```
while(true) {
    // action – durée 1ms
    Thread.sleep(4);
}
```



Besoins en facultés temporelles

- Interface avec les notions liées au temps
  - accéder à des horloges pour mesurer le passage du temps
  - retarder un processus jusqu'à un moment ultérieur
  - programmer une temporisation (*time-out*) pour traiter l'absence d'un événement
- Représentation des exigences temporelles
  - spécification des périodes des tâches
  - spécification des échéances (*deadline*)

Temporisation des synchronisations

- Besoin: limiter le temps qu'une tâche peut être bloquée en attente d'une communication / synchronisation

  Exemples:

  - si les données du capteur de température ne sont pas disponibles au bout de 100ms, signaler la panne
  - si le consommateur ne libère pas une place dans le buffer au bout de 100ms, annuler l'écriture en cours

⇒ ces temporisations sont liées aux primitives de synchronisation

Exemple POSIX

```
if(sem_timedwait(&sem, &time) < 0) {
        if(errno == ETIMEDOUT)
                … // temps expiré
} else {
        … // sémaphore bloqué
}
```

Exemple Java

```
RelativeTime delta=…;
AbsoluteTime t0;
Clock rtc = Clock.getRealtimeClock();
…
t0 =rtc.getTime();
while(!cond) {
      HighResolutionTime.waitForObject(this, delta)
      if(rtc.getTime().substract(t0).compareTo(delta) >= 0) {
            // cause : temps expiré
      } else {
            // cause : notify
      }
}
```

Besoins en facultés temporelles

- Interface avec les notions liées au temps
  - accéder à des horloges pour mesurer le passage du temps
  - retarder un processus jusqu'à un moment ultérieur
  - programmer une temporisation (*time-out*) pour traiter l'absence d'un événement

- Représentation des exigences temporelles
  - spécification des périodes des tâches
  - spécification des échéances (*deadline*)
  - ...

Besoins

- Identifier les éléments à caractériser
  - un bloc de code
  - une fonction
  - une tâche
  - un scénario impliquant plusieurs tâches (contraintes *de-bout-en-bout*)
- Donner les caractéristiques gros-grain d'exécution
  - arrivée *périodique*, *apériodique*, *sporadique*
- Caractériser finement l'aspect temporel
  - période ou temps minimum entre arrivées
  - temps d'exécution au pire cas
  - échéance relative ou absolue
  - temps d'attente maximum pour une ressource
  - …
- Associer des actions (handler) pour le cas où les spécifications ne sont pas respectées

La réalité

Ce type d'information est supporté dans

- peu de langages **de programmation**
- de nombreux langages de **spécification** et **modélisation**
  - mais sans support réel pour l'implémentation

Une exception (qui confirme la règle) : RT Java

```
class javax.realtime.RealtimeThread {
  RealtimeThread(SchedulingParameters scheduling,
                 ReleaseParameters release,
                 MemoryParameters memory,
                 MemoryArea area,
                 ProcessingGroupParameters group,
                 java.lang.Runnable logic)
  … }
```

Facultés temporelles – résumé

- besoins :
  - accès à des horloges
  - retarder une tâche
  - temporisations
  - spécification des caractéristiques temporelles des tâches
- tolérance aux fautes temporelles :
  - détection du dépassement d'échéance
  - détection du dépassement de temps CPU
  - détection du non-respect des lois d'arrivée
  - …