Exemple : création et activation d'une tâche (one-shot)

Dans cet exemple il s'agit de créer et de lancer une tâche qui exécute une séquence simple d'instructions. La tâche est ensuite détruite naturellement quand le pme ppal qui l'a crée se termine. Par exemple, créer une tâche de forte priorité qui affiche « hello World ».
Créer la tâche dans le programme principal avec le mode T_JOINABLE et attendre que la tâche ait terminé son exécution avec la fonction rt_task_join(&task_desc).
Utiliser la fonction rt_task_spawn () pour créer et activer la tâche.

Penser à récupérer systématiquement le code retourné par la fonction de création de la tâche afin de s'assurer que cela s'est bien passé.

Rque : créer une tâche Xenomai qui affiche un message n'a aucun intérêt d'un point de vue temps réel mais permet simplement de présenter la structure d'une application à travers un exemple basique.

Exemple (variante) : création et activation d'une tâche avec passage de paramètre à l'appel

Reprenons l'exemple précédent avec une variante : au lieu de prédéfinir la chaîne de caractère à afficher dans la tâche ('Hello World'), on la lui passe en argument.

On applique les bonnes pratiques de programmation et on note symboliquement les arguments des fonctions.

Exemple : création, activation et destruction d'une tâche

Dans cet exemple il s'agit de créer et de lancer une tâche qui s'exécute mais ne se termine pas, puis de la détruire quand l'utilisateur le souhaite. Par exemple, cette tâche, de forte priorité, affiche « hello World » (voir premier exemple) puis se met en sommeil indéfiniment (rt_task_sleep_until(TM_INFINITE)).

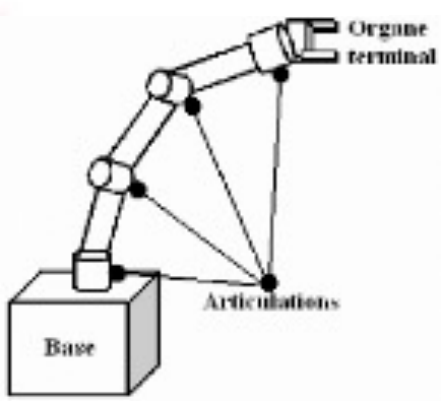
NB : Utiliser pour changer les fonctions rt_task_create () et rt_task_start () pour respectivement créer et activer la tâche.
Penser à récupérer systématiquement le code retourné par les fonctions.
On applique toujours les bonnes pratiques de programmation et on note symboliquement les arguments des fonctions.

Définir spécifiquement une fonction de destruction de la tâche en sommeil qui sera appelée par l'utilisateur.

Exemple : création, activation et destruction de plusieurs tâches – illustration de la réentrance

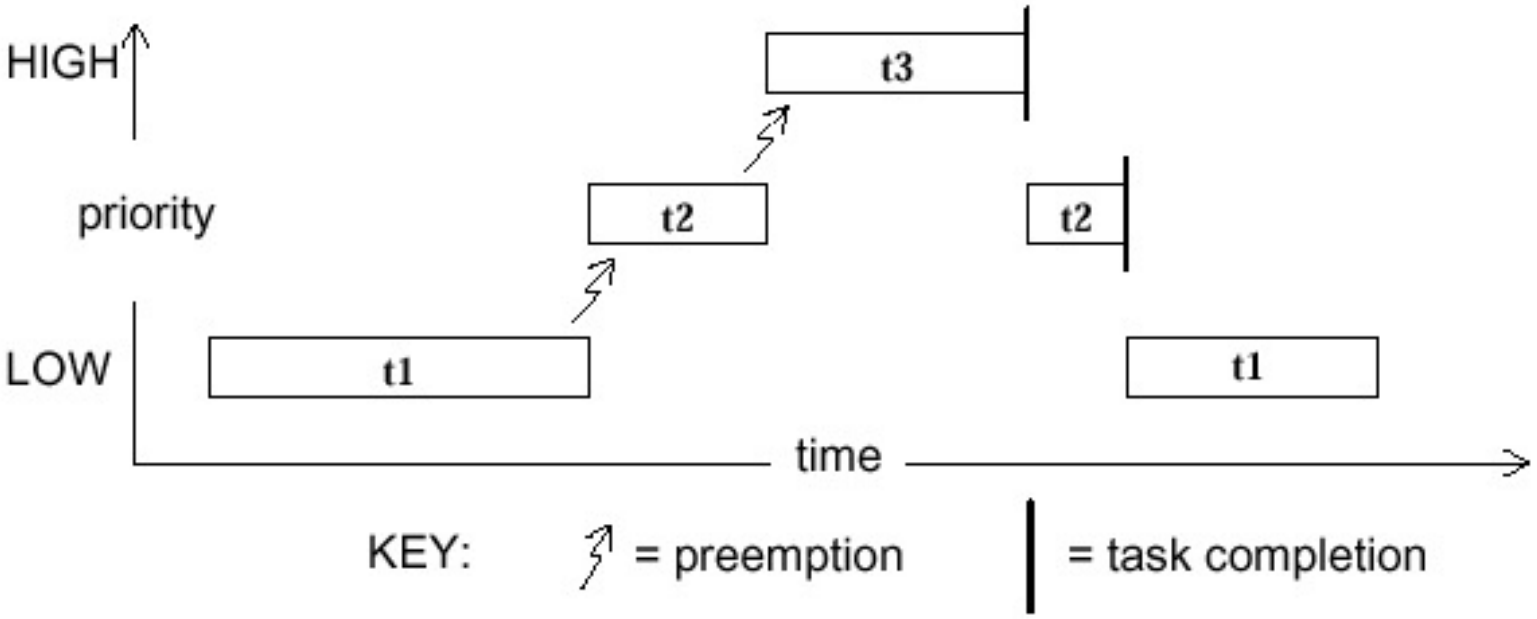
Créer trois tâches de même priorité, « poignet », « coude » et « épaule » qui vont piloter en temps réel et de façon autonome et indépendante le déplacement d'un bras de robot anthropomorphe. Le mouvement du bras résulte de la coordination fine des mouvements des trois articulations.

Coude, épaule et poignet fonctionnent sur le même principe.

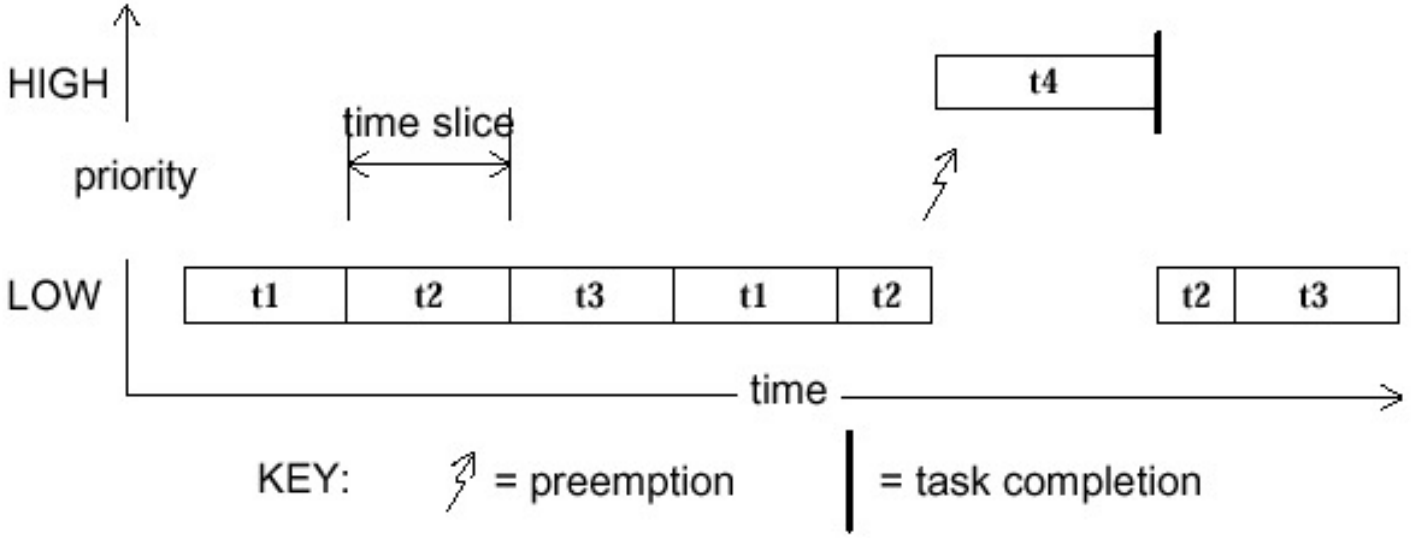


NB :
Utiliser la fonction rt_task_spawn () pour créer et activer les tâches.
Utiliser la terminaison adoptée dans le TP.

Ordonnancement préemptif basé priorité



Avec Round Robin



Exemple : créer et activer une tâche périodique

Reprendre l’exemple du bras de robot et rajouter un capteur de position sur chaque articulation, qui donne à chaque seconde la position angulaire de l’articulation.

Exemple : créer et activer une tâche périodique

Variante incluant la détection de l'overrun de la tâche périodique, c'est-à-dire le nombre de fois où le réveil de la tâche a échoué car elle était encore en cours d'exécution.

Mutex

- Le mutex est un mécanisme simple de verrou permettant la protection d'une section critique.
- En anglais: *mutex*, pour mutual exclusion
  - Lorsqu'une tâche désire accéder la section critique, elle doit d'abord acquérir le verrou. Celui-ci doit être ouvert, c-à-d qu'aucune autre tâche se trouve dans la section critique.
    - La tâche devient (momentanément) propriétaire du mutex.
  - La tâche libère le verrou en fin de traitement; il redevient ouvert.
  - Si le verrou est fermé lors de l'acquisition par une tâche, cette dernière est suspendue jusqu'à ce que le verrou redevienne ouvert.
- Le mutex est un mécanisme d'attente passive.
  - Basé sur le principe qu'il est inutile d'activer une tâche en attente de la section critique.
  - La libération de la section critique est mise à profit pour relancer une des tâches en attente.

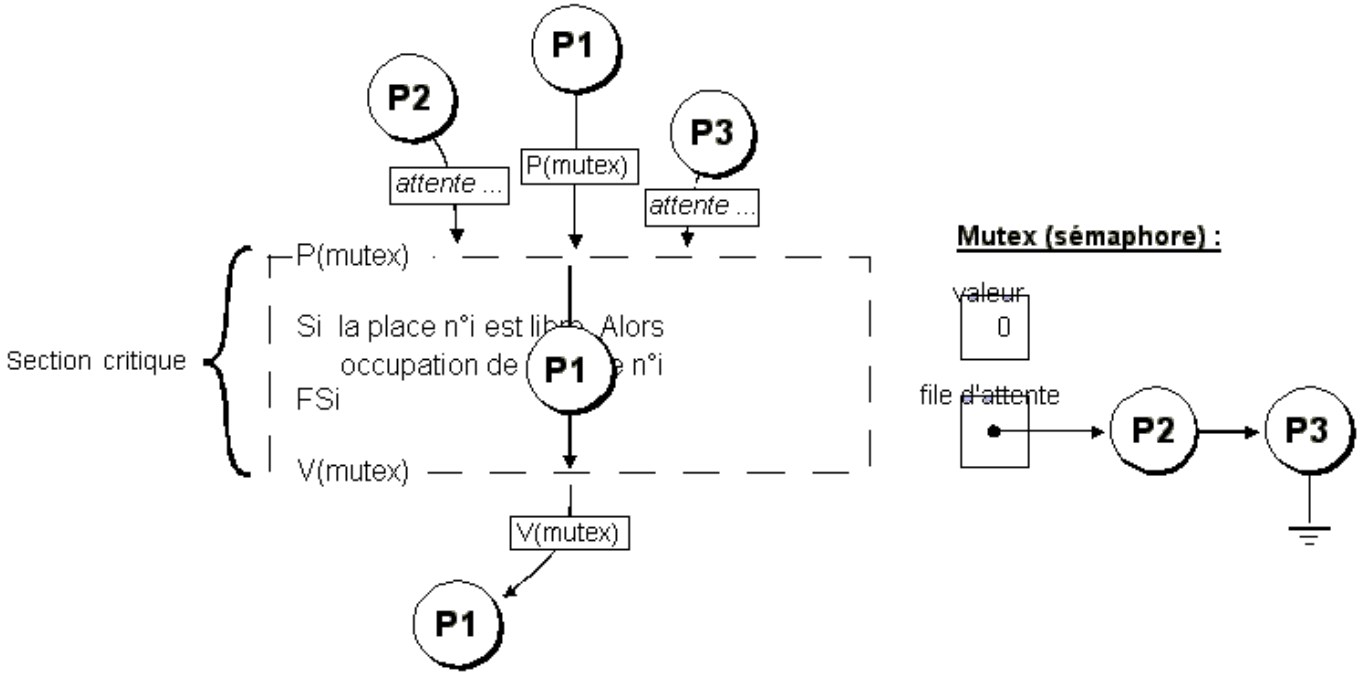
Mutex

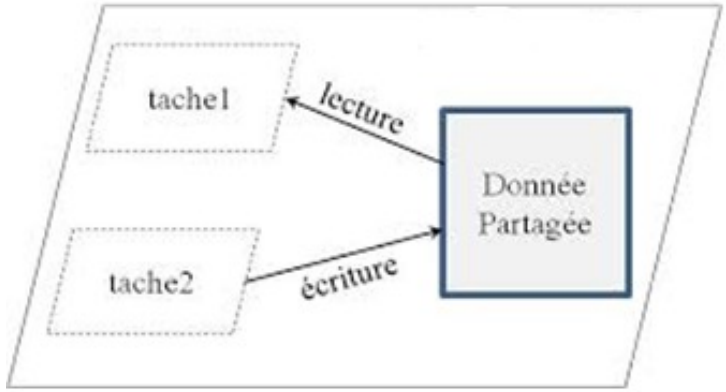
- Le Mutex
  - est une variable booléenne
  - possède une liste d'attente
  - est manipulé par deux opérations *atomiques*:
    - `Verrouille(v)`

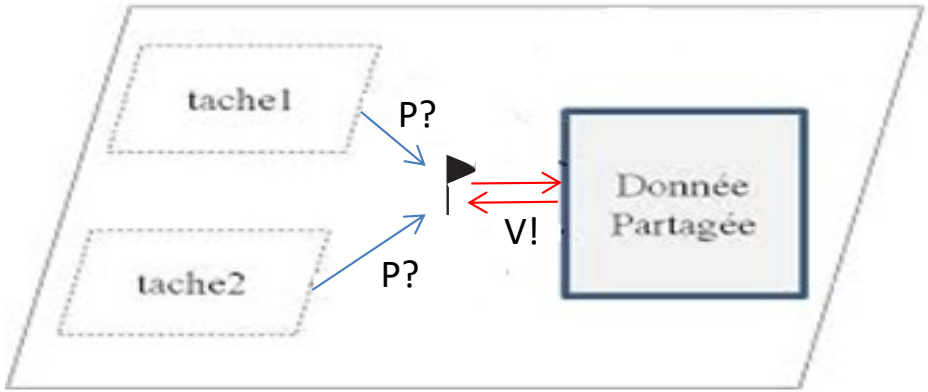
```
void Verrouille(verrou v)
{
  if (v)
    v = false;
  else
    suspendre la tâche appelante dans la file associée à v
}
```

    - `Deverrouille(v)`

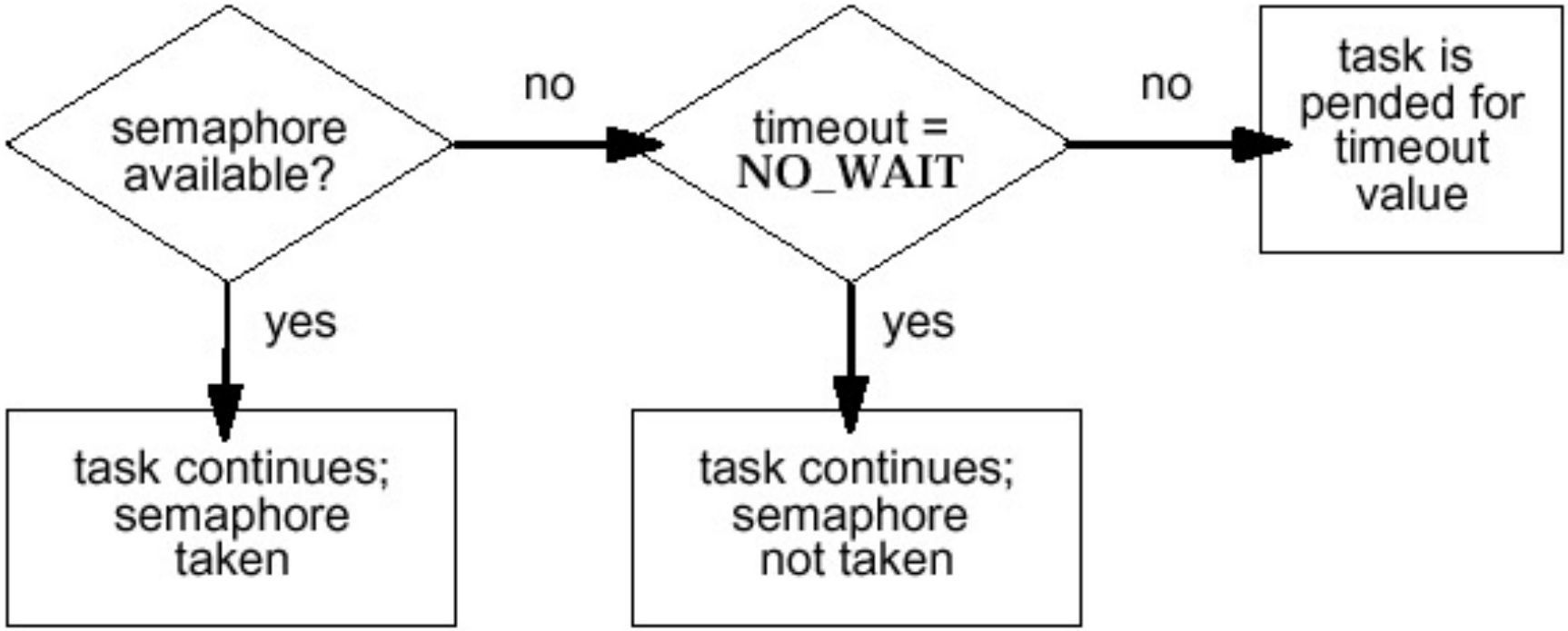
```
void Déverrouille(verrou v)
{
  if (la file associée à v != vide)
    débloquer une tâche en attente dans file
  else
    v = true;
}
```

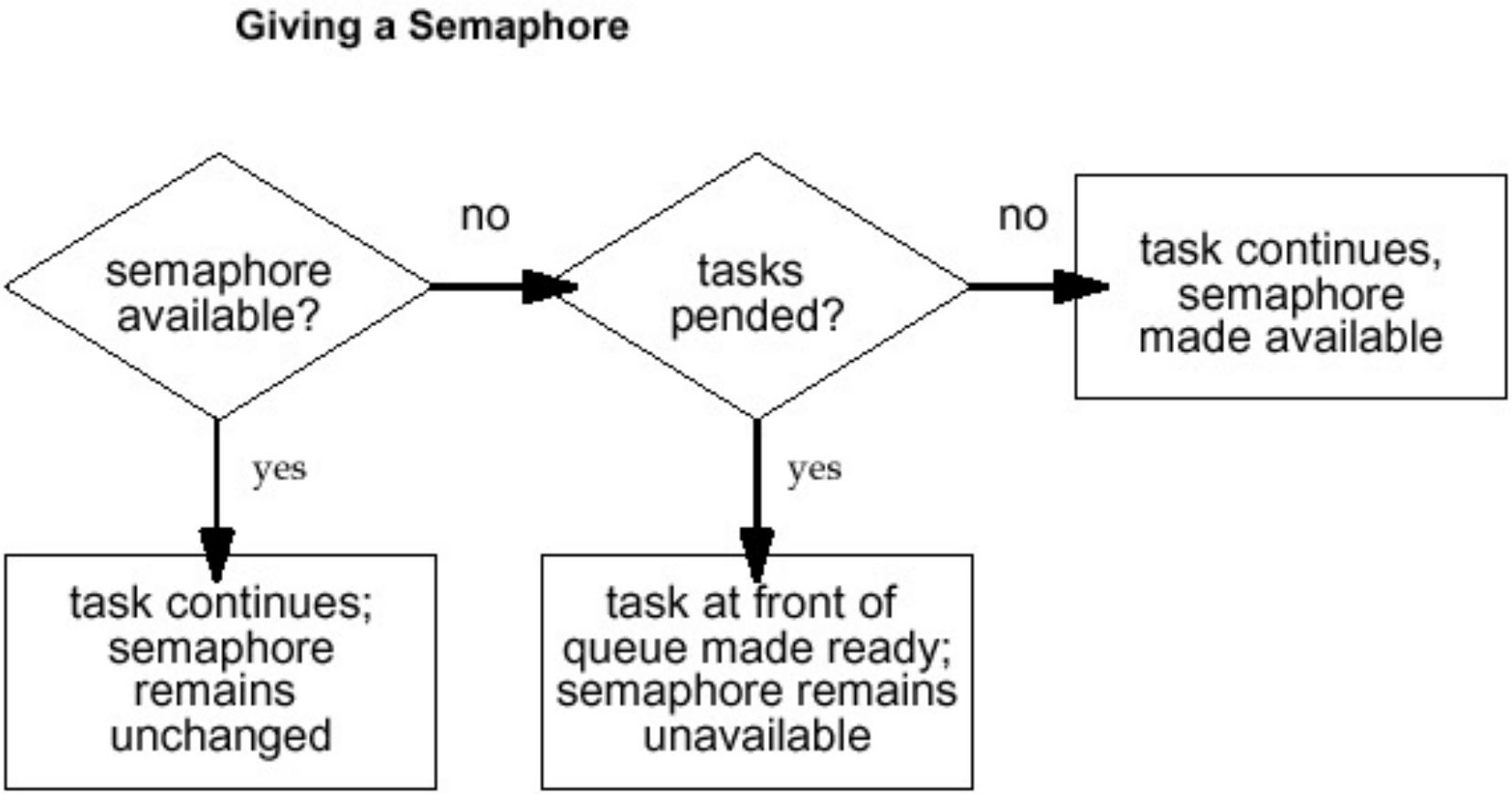

P1
P2
P3
P(mutex)
attente ...
attente ...
Section critique
P(mutex)
Si la place n°i est libre Alors
occupation de la place n°i
FSi
V(mutex)
V(mutex)
P1
Mutex (sémaphore) :
valeur
0
file d'attente
P2
P3


tache1
lecture
Donnée
Partagée
tache2
écriture


tache1
P?
Donnée
Partagée
V!
tache2
P?

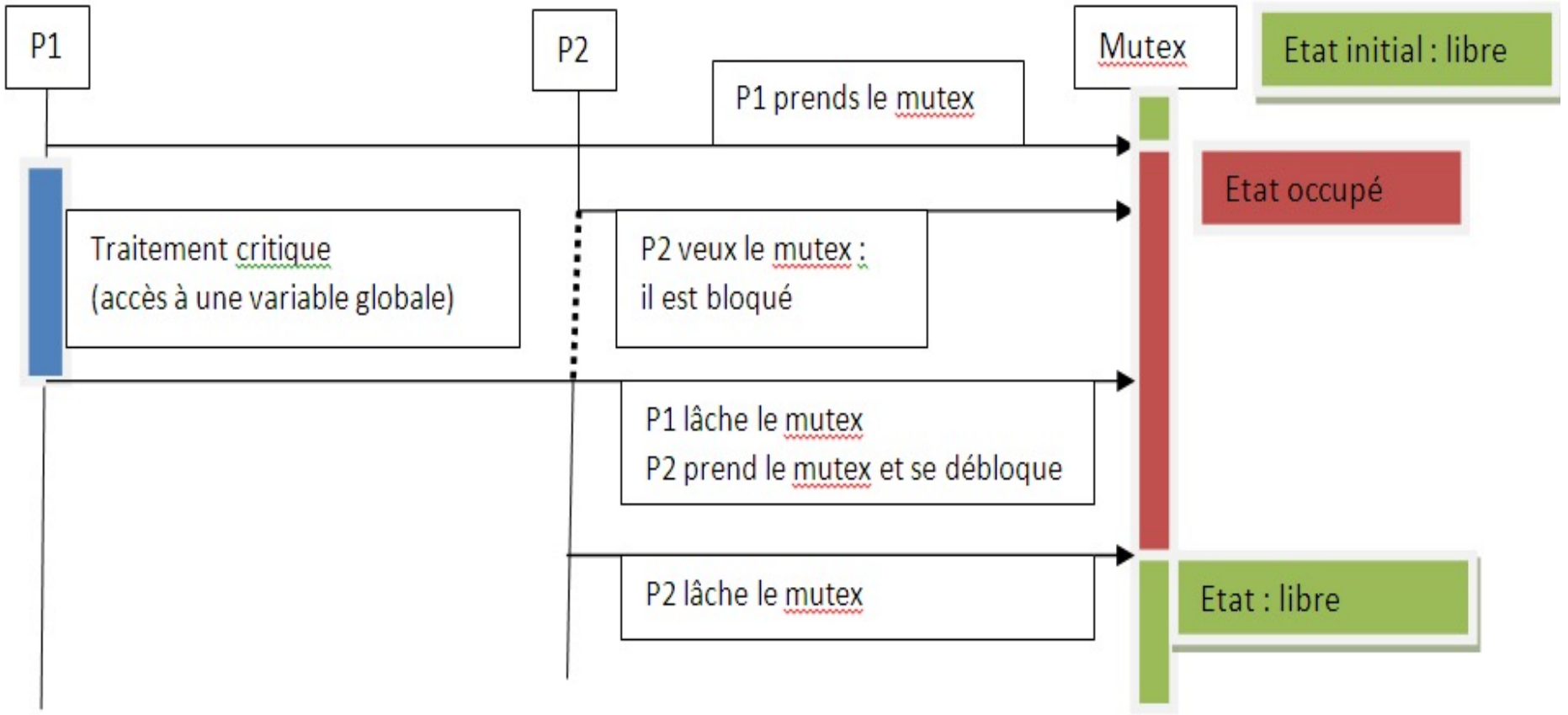
Taking a Semaphore



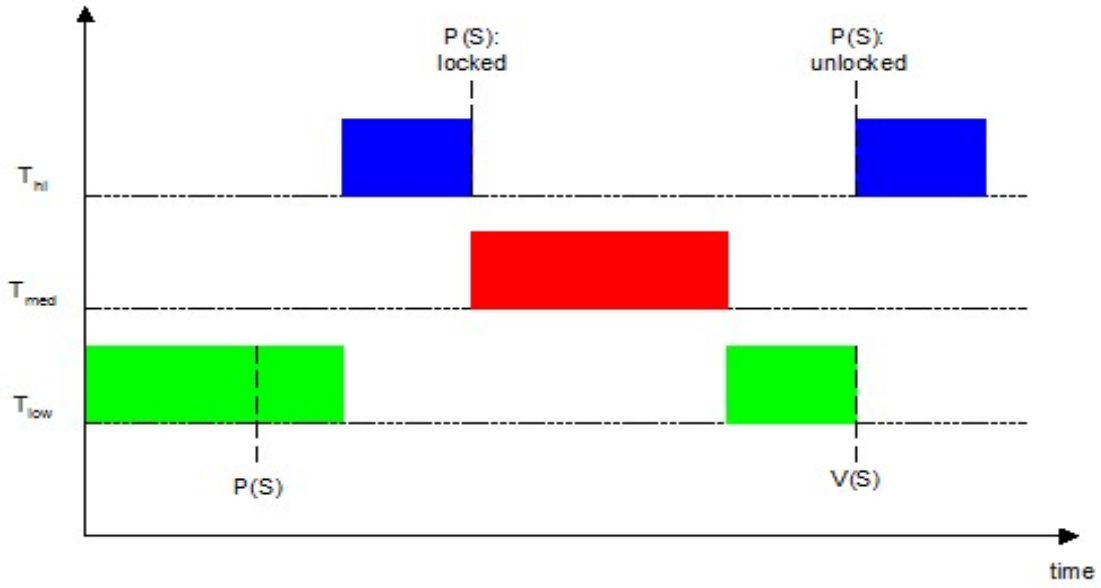


Flush ??

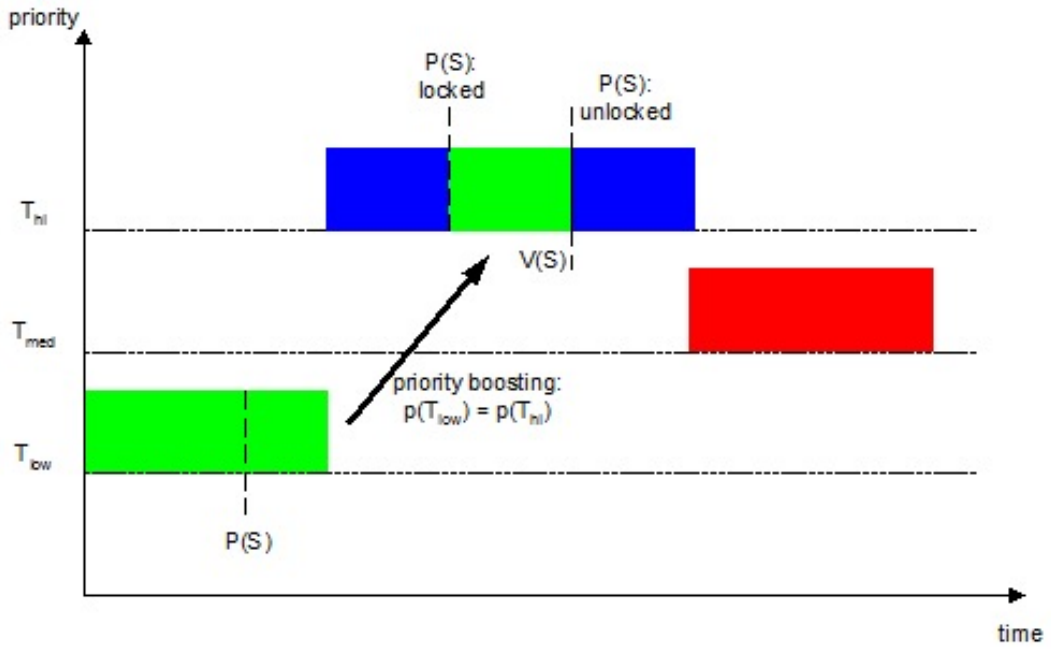
Protection en destruction ???


P1
P2
Mutex
Etat initial : libre
P1 prends le mutex
Etat occupé
Traitement critique
(accès à une variable globale)
P2 veux le mutex :
il est bloqué
P1 lâche le mutex
P2 prend le mutex et se débloque
P2 lâche le mutex
Etat : libre

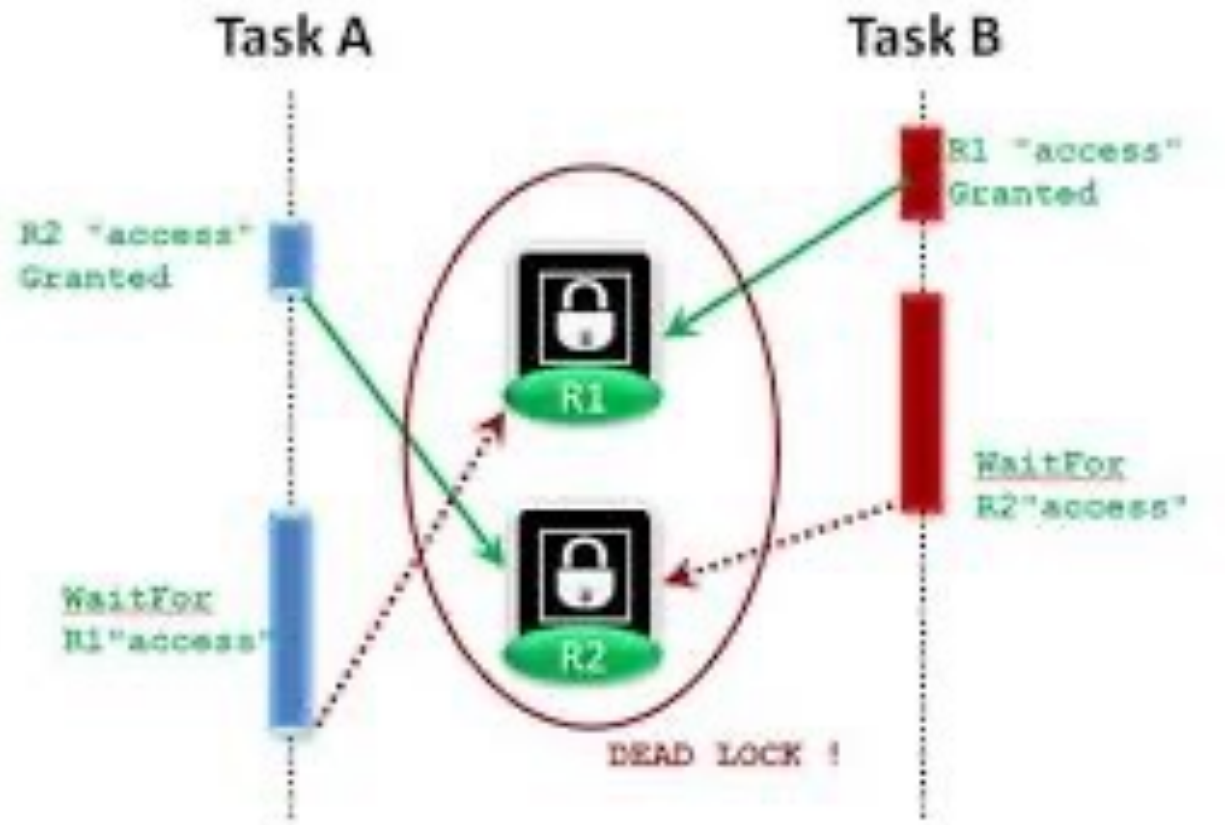
Problème : inversion de priorité



Solution : héritage de priorité



Interblocage

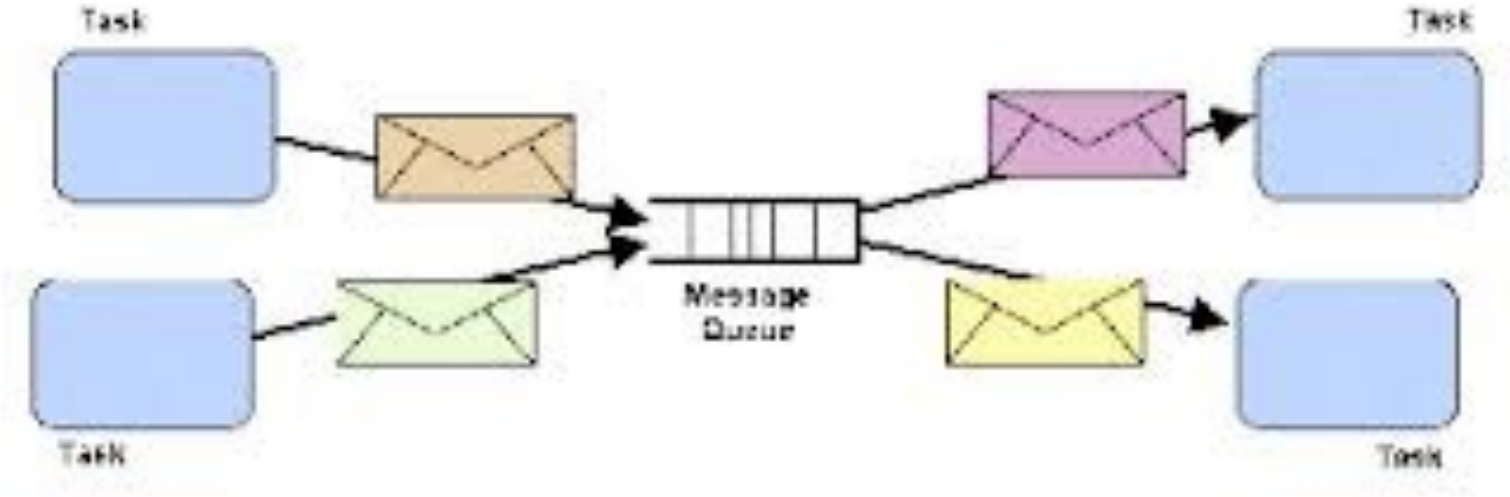


Exemple : utilisation d’un mutex

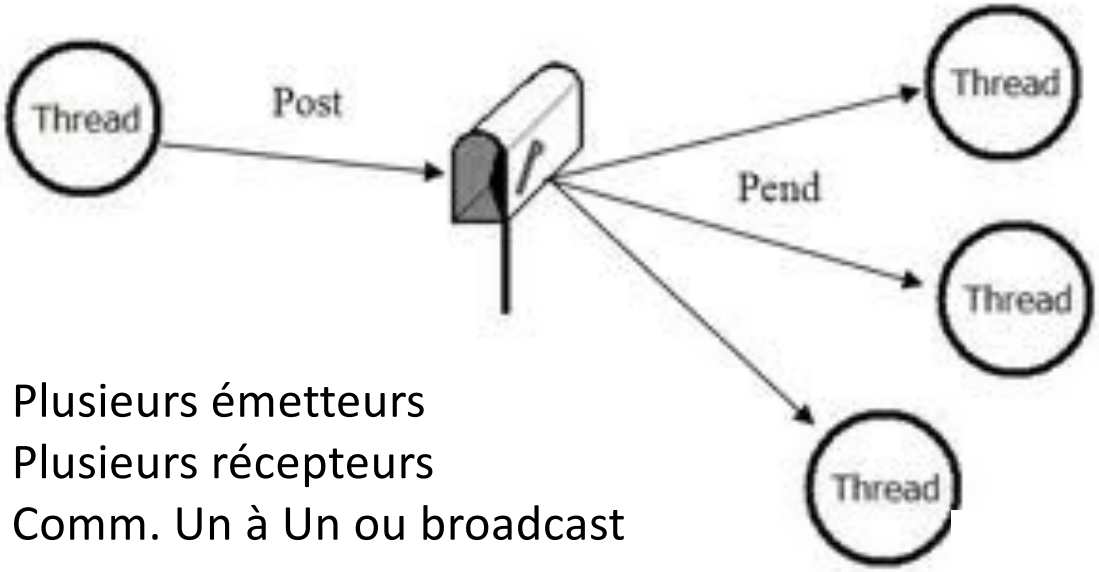
Reprendre l’exemple du bras de robot et rajouter la protection des variables de position des articulations

Communication par file de messages

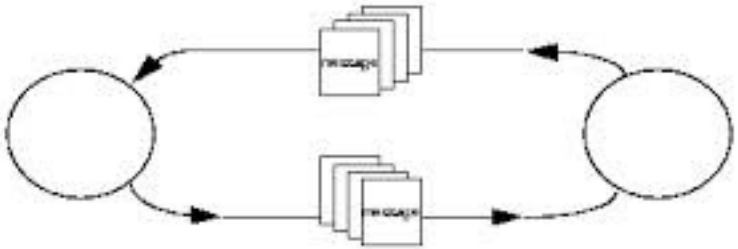
Absorption différence rythme
Stockage des données sans écrasement
Synchronisation
Exclusion mutuelle
Priorisation des messages à l'envoi
Gestion des émetteurs/récepteurs en attente



Exemple : mailbox sous Gmail

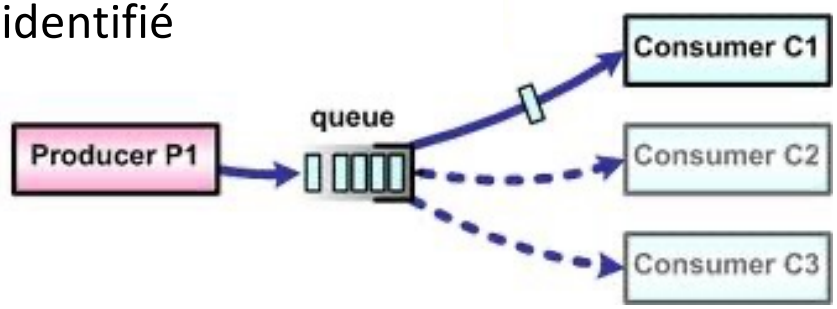


Plusieurs émetteurs
Plusieurs récepteurs
Comm. Un à Un ou broadcast

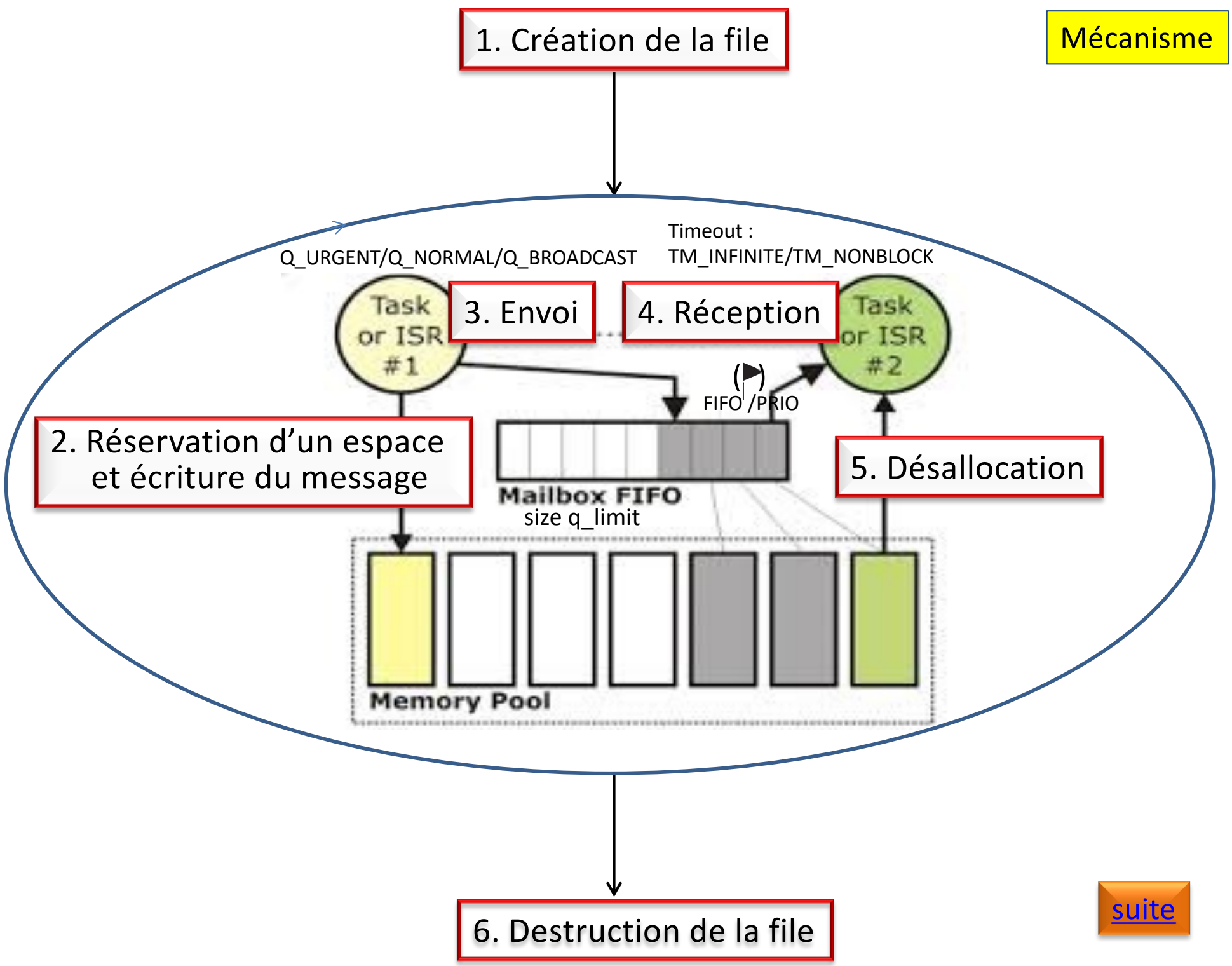


Unidirectionnel …

Pas de destinataire identifié



Mécanisme



suite

Création de la file

```
int rt_queue_create ( RT_QUEUE* q,
                      const char * name,
                      size_t poolsize,
                      size_t qlimit,
                      int mode );
```

address of a queue descriptor

ASCII string - symbolic name

size (bytes) of the message buffer pool - fixe

maximum number of messages / Q_UNLIMITED

mode :

- Q_FIFO makes tasks **pend** in FIFO order on the queue for **consuming** messages.
- Q_PRIO makes tasks pend in priority order on the queue.

Create a message queue object that allows multiple tasks to exchange data through the use of variable-sized messages.
A message queue is created empty.

retour

Réservation d'un espace puis écriture du message

```
void  rt_queue_alloc ( RT_QUEUE* q,
                       size_t size );
```

address of a queue descriptor

requested size in bytes of the buffer

Allocate a message queue buffer from the queue's internal pool which can be subsequently filled by the caller (memcpy() ou strcopy()) then passed to rt_queue_send() for sending.

retour

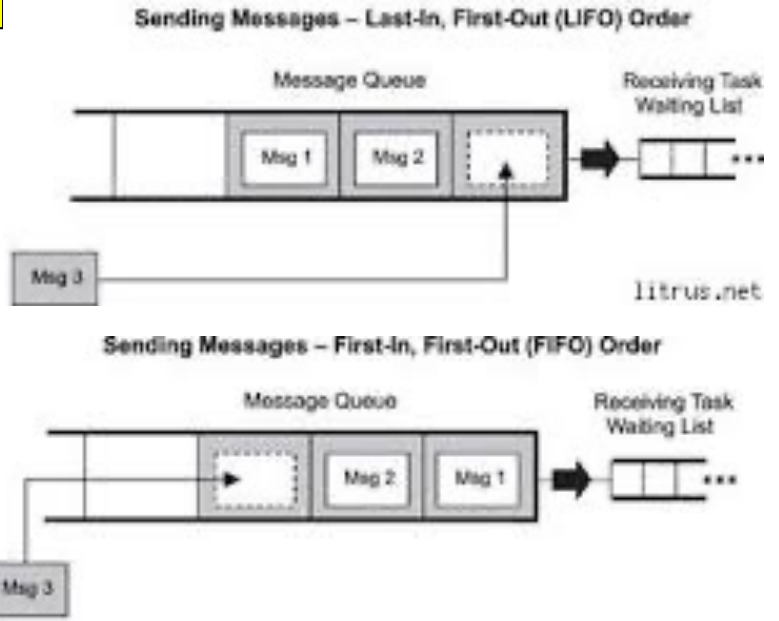
Envoi

```
int rt_queue_send (   RT_QUEUE* q,
                      void *mbuf,
                      size_t size,
                      int mode  );
```

address of a queue descriptor

address of the message buffer to be sent*

size in bytes of the message
(0: an empty message will be sent)



mode :

- Q_URGENT : message prepended to the message queue (LIFO ordering)
- Q_NORMAL : message appended to the message queue (FIFO ordering)
- Q_BROADCAST : message sent to all tasks currently waiting for messages.

Send a message to a queue.

The message must have been allocated by a previous call to rt_queue_alloc().

Returns the number of receivers which got awaken as a result of the operation

retour

Réception

```
ssize_t  rt_queue_receive (   RT_QUEUE* q,
                              void **bufp,
                              RTIME timeout  );
```

pointer to a memory location which will be written upon success with the address of the received message.

Timeout : number of clock ticks to wait for a message to arrive.

- TM_INFINITE : caller blocked indefinitely until some message is eventually available
- TM_NONBLOCK : no waiting if no message is available

Receive a message (next available) from a queue.
Unless otherwise specified, the caller is blocked for a given amount of time if no message is immediately available on entry.
The number of bytes available from the received message is returned upon success

retour

Désallocation

Once consumed, the message space should be freed using rt_queue_free().

```
void  rt_queue_free ( RT_QUEUE* q,
                      void *buf );
```

Releases a message buffer returned by rt_queue_receive() to the queue's internal pool.

retour

Destruction de la file

int rt_queue_delete (RT_QUEUE* q);

address of a queue descriptor

Delete a message queue.
Destroy a message queue and release all the tasks currently pending on it.
A queue exists in the system since rt_queue_create() has been called to create it, so this service must be called in order to destroy it afterwards.

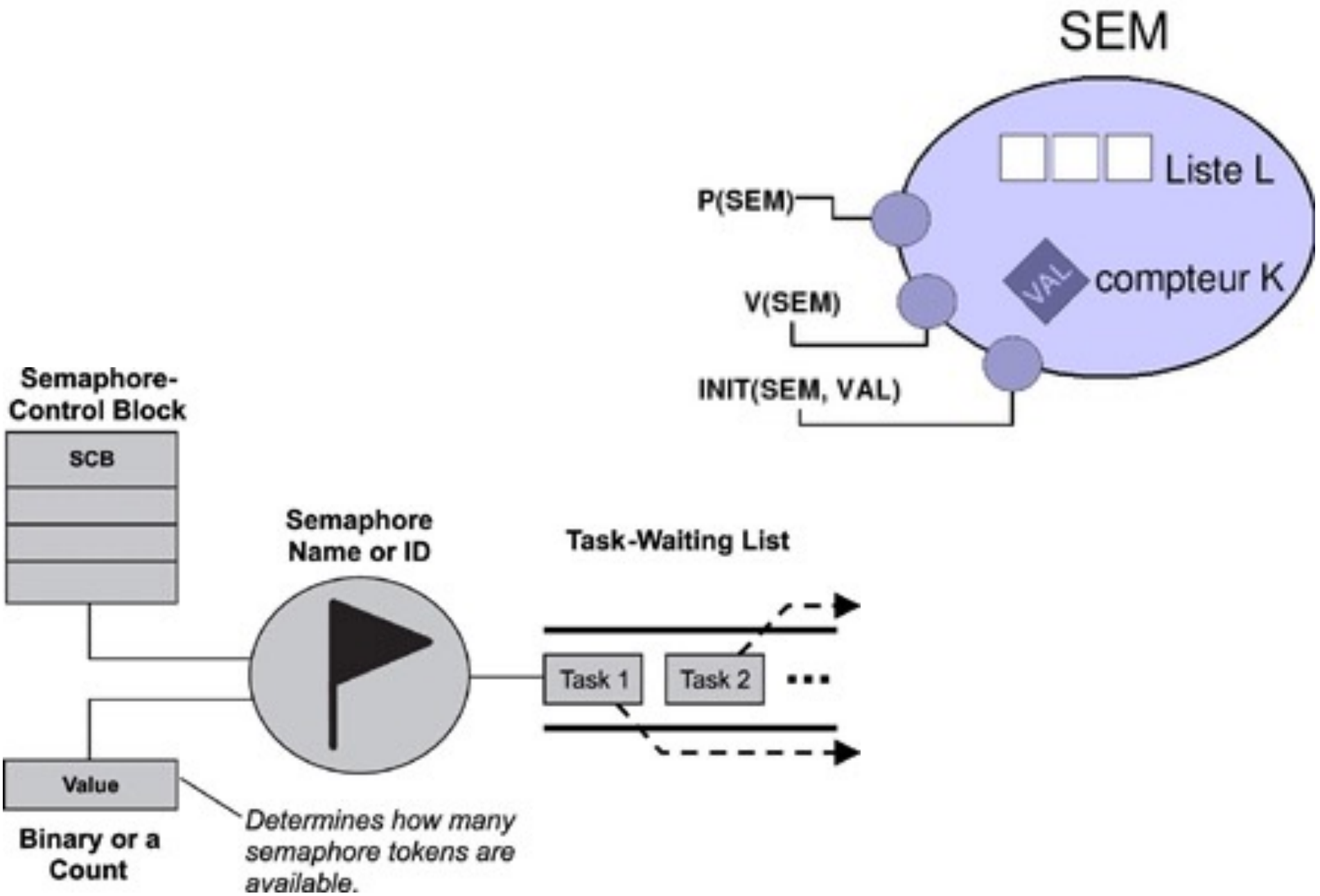
retour

Exemple : deux tâches s'envoient en boucle des messages par une file de message

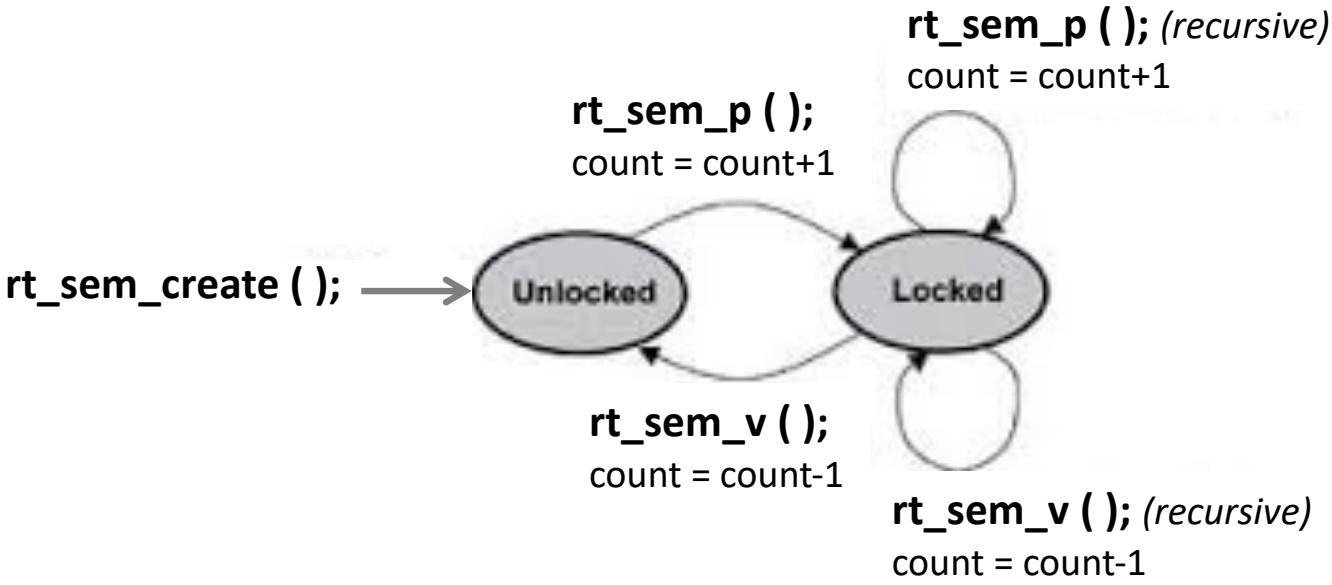
Jouez avec les différents paramètres :

- Priorités et ordonnancement
- Taille de la file et timeouts
- Gestion
- Tailles des messages R/W

Sémaphores à compteur : construction

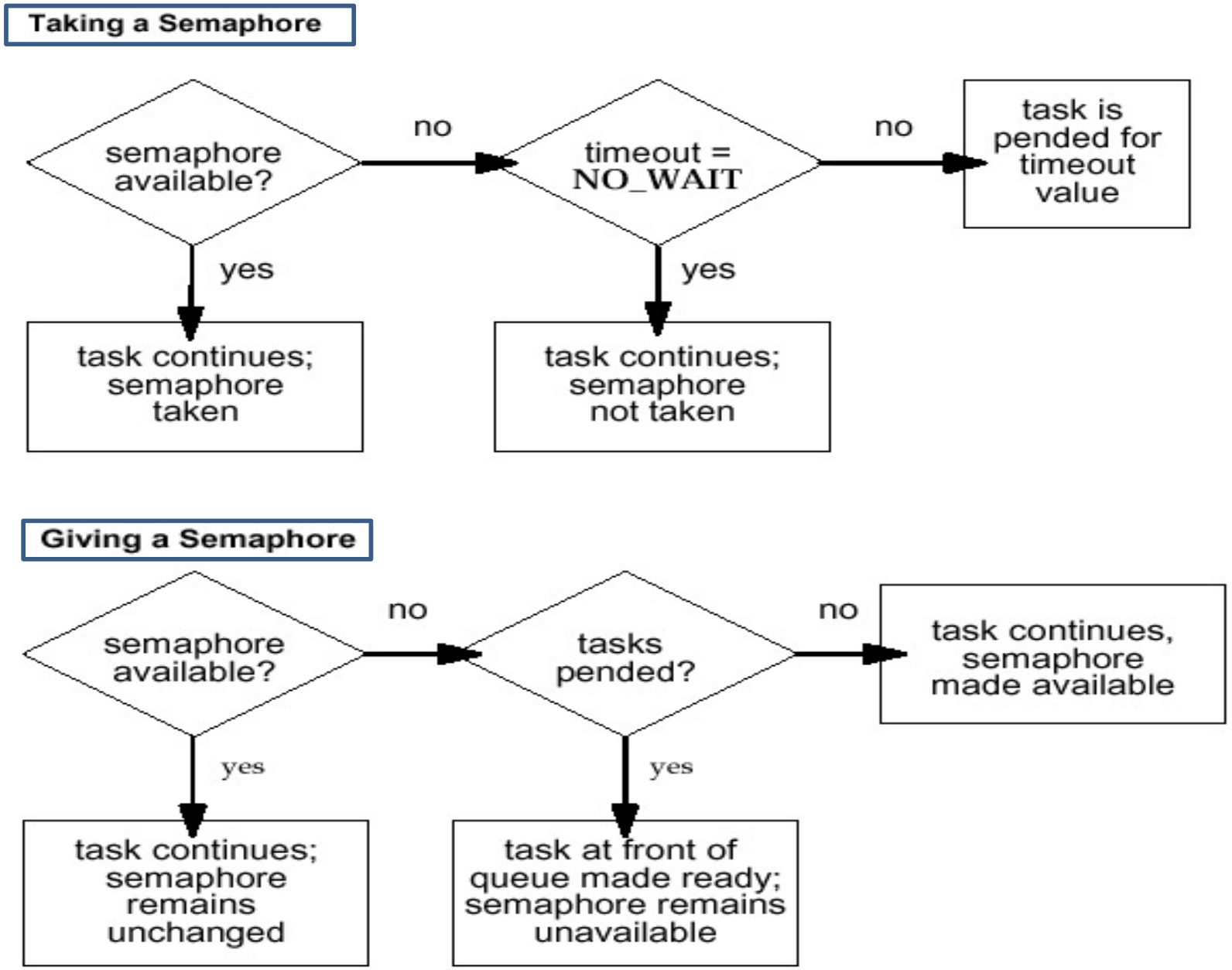


Sémaphores à compteur : fonctionnement

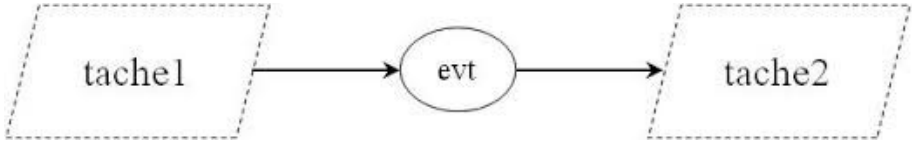


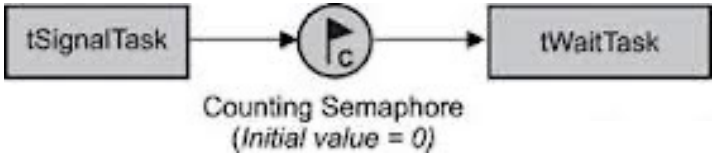
Semaphore Call	Count after Call	Resulting Behavior
rt_sem_create ( );	3	Semaphore initialized with initial count of 3.
rt_sem_p ( )	2	Semaphore taken.
rt_sem_p ( )	1	Semaphore taken.
rt_sem_p ( )	0	Semaphore taken.
rt_sem_p ( )	0	Task blocks waiting for semaphore to be available.
rt_sem_v ( )	0	Task waiting is given semaphore.
rt_sem_v ( )	1	No task waiting for semaphore; count incremented.

Sémaphores à compteur : opérations



Application 1 : Synchronisation par sémaphores

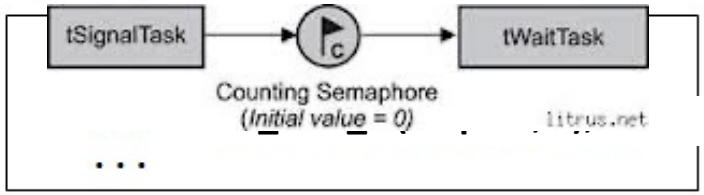





```
DispSem   =   rt_sem_create(…);
```

Semaphore created "empty"

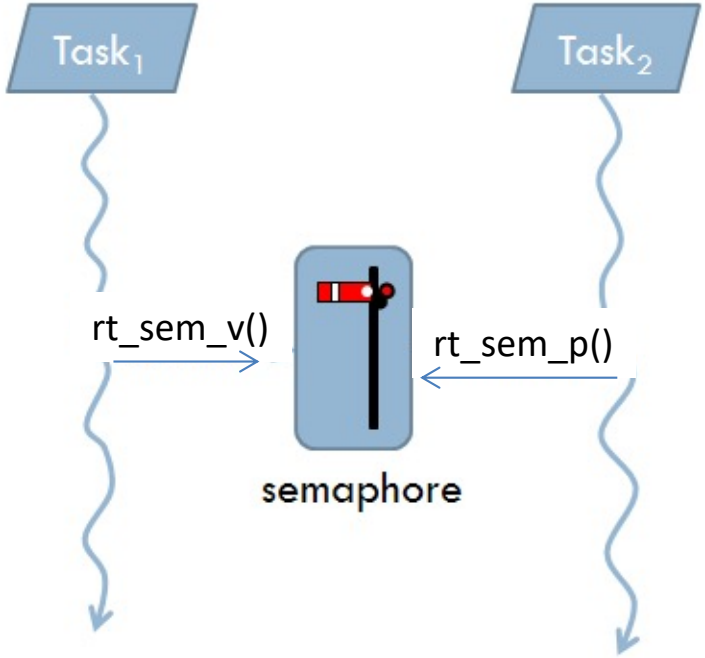
Task$_1$




```
...
```

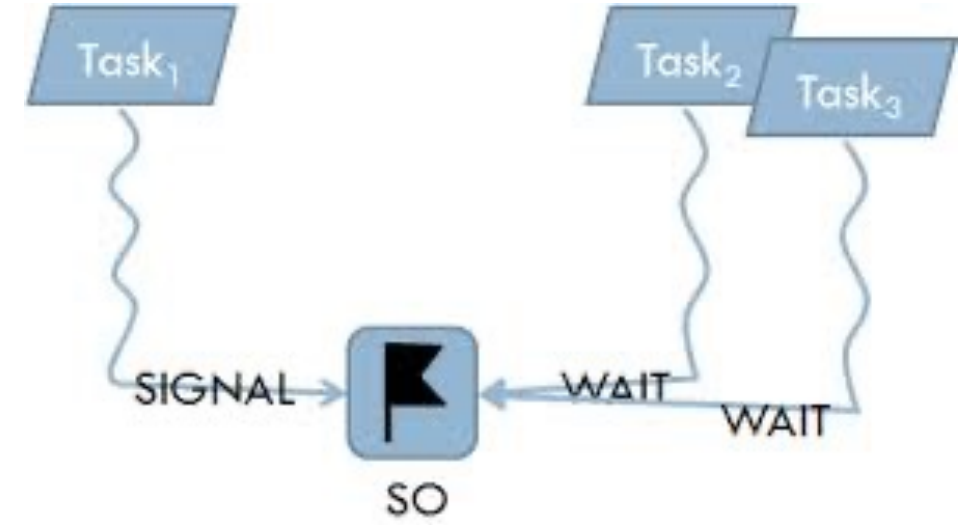
Task$_2$

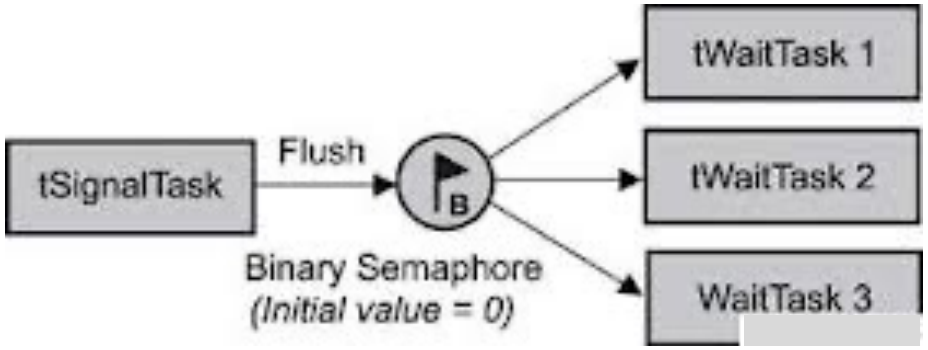
```
...
/* WAIT */
 rt_sem_p (Dispem…);
...
```

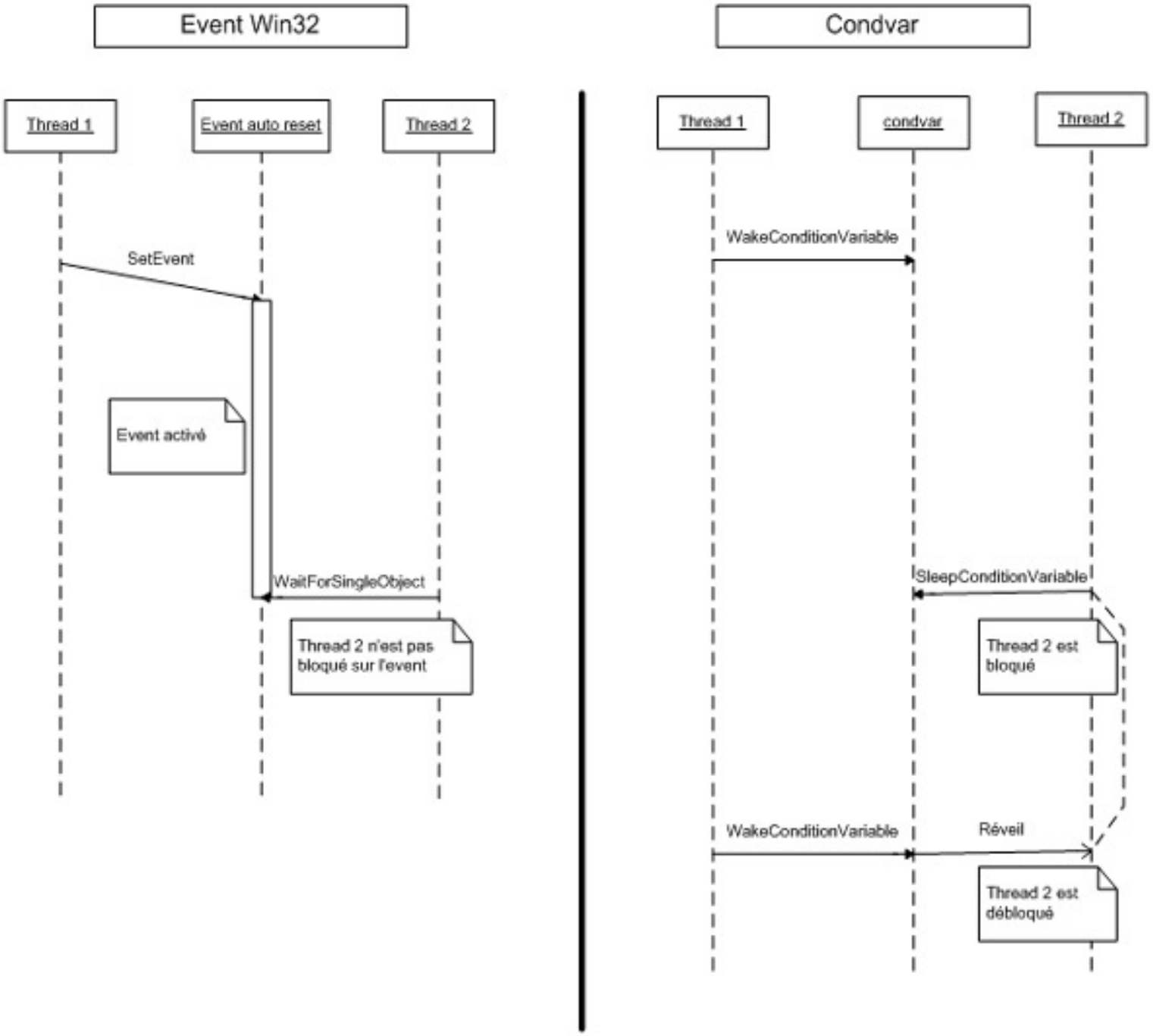


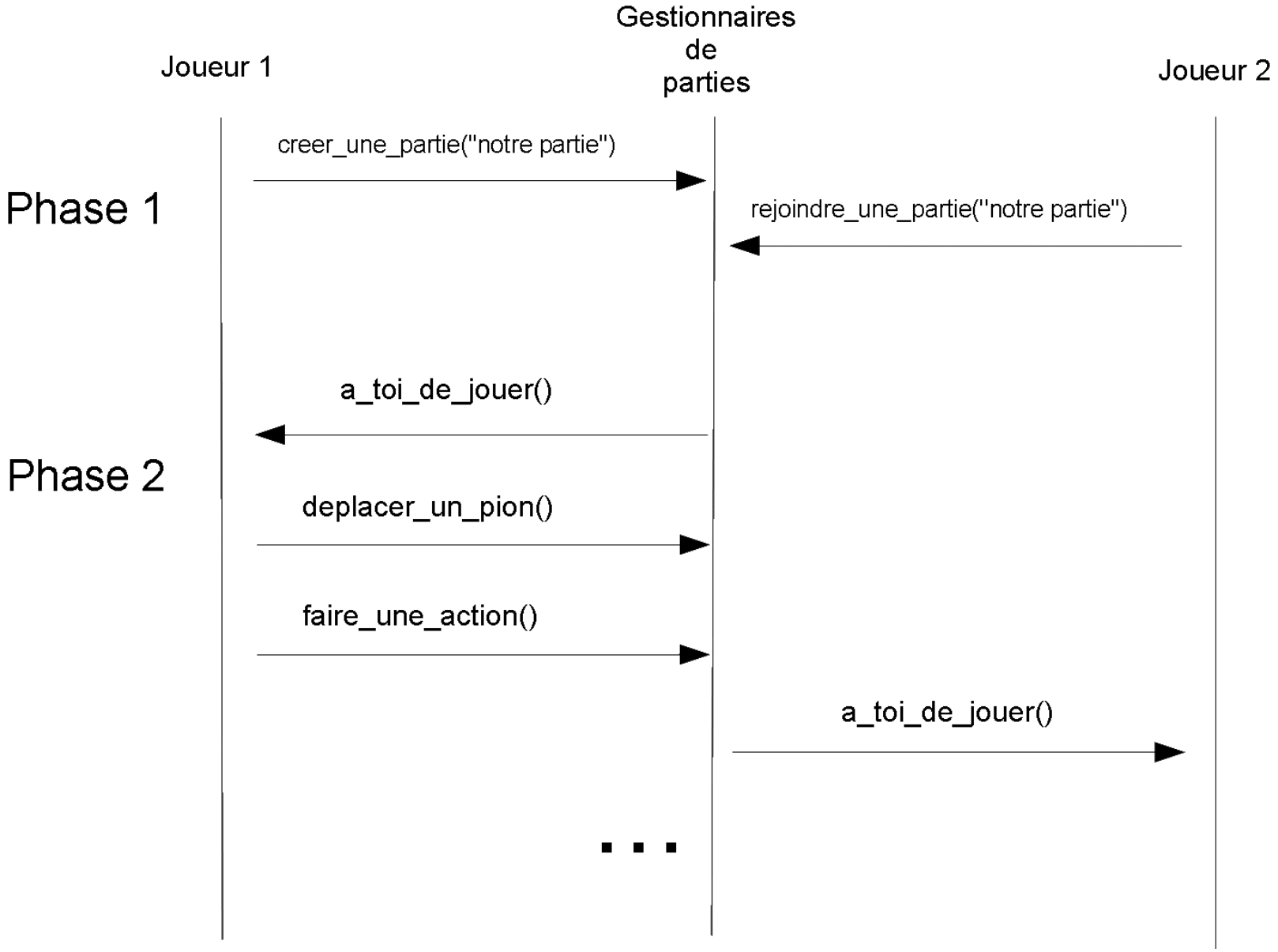
Application 1 : Synchronisation par sémaphores

Synchronisation multiple :

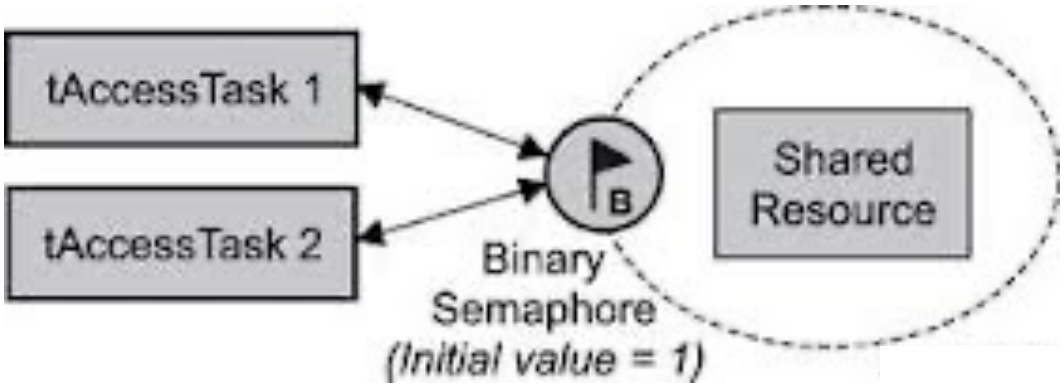




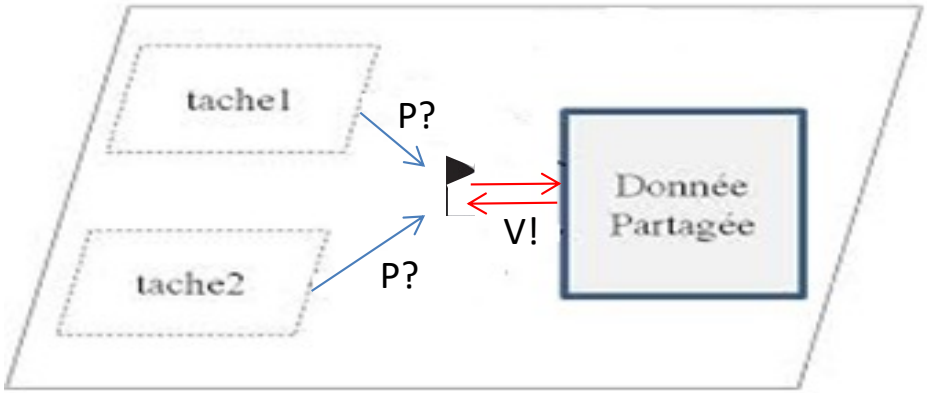

Event Win32
Thread 1
Event auto reset
Thread 2
SetEvent
Event activé
WaitForSingleObject
Thread 2 n'est pas bloqué sur l'event
Condvar
Thread 1
condvar
Thread 2
WakeConditionVariable
SleepConditionVariable
Thread 2 est bloqué
WakeConditionVariable
Réveil
Thread 2 est débloqué


Joueur 1
Gestionnaires de parties
Joueur 2
Phase 1
creer_une_partie("notre partie")
rejoindre_une_partie("notre partie")
Phase 2
a_toi_de_jouer()
deplacer_un_pion()
faire_une_action()
a_toi_de_jouer()
. . .

Application 2 : Exclusion mutuelle par sémaphore à compteur



Même mécanisme qu'avec un mutex ! (options différentes…)



Exemple d'exclusion mutuelle

```
RT_SEM sem_desc;

void  T1 (void *arg) {
            rt_sem_p(&sem_desc, TM_INFINITE);
            // section partagée
            rt_sem_v(&sem_desc);
}
void  T2 (void *arg) {
            rt_sem_p(&sem, TM_INFINITE);
            // section partagée
            rt_sem_v(&sem);
}
void  T3 (void *arg) {
            rt_sem_p(&sem_desc, TM_INFINITE);
            // section partagée
            rt_sem_v(&sem_desc)
}
void main() {
            // init. du sémaphore à SEM_INIT =1
            rt_sem_create (&sem_desc,"Semaphore", SEM_INIT, S_PRIO);
            // création des taches
}
```

Alarmes : Watchdog timers

Alarms can be either one shot or periodic; in the latter case, the real-time kernel automatically reprograms the alarm for the next shot according to a user-defined interval value.

Periodic Execution

```
rt_alarm_create (&alarm_desc, “my_alarm”);
rt_alarm_start (&alarm_desc, start_time, period);
...
void treat_alarm (int param)
   {
    while (1)
        {/* Wait for the alarm expiration */
        rt_alarm_wait(&alarm_desc);
         /* Process the work */
        doIt (.....);}
   }
rt_alarm_delete (&alarm_desc);
```

Alarms can be made periodic or oneshot, depending on the reload interval

Basic principle:
define some alarm server task which routinely waits for the next incoming alarm event

Deadline-miss Recovery

```
rt_alarm_create (&alarm_desc, “my_alarm”);
void Critical_periodic_task (int param)
{
while (1)
   {
   /* wait next release (periodic task or semaphore) */
   rt_alarm_start (&alarm_desc, start_time, period);
   doWork();
   rt_alarm_stop (&alarm_desc);
   }
}
void Deadline_detection ()
{
   while (1)
  {
   rt_alarm_wait(&alarm_desc);
    /* Handle deadline miss */
   }
}
```

Pour désarmer une alarme :
`rt_alarm_stop (&alarm_desc);`