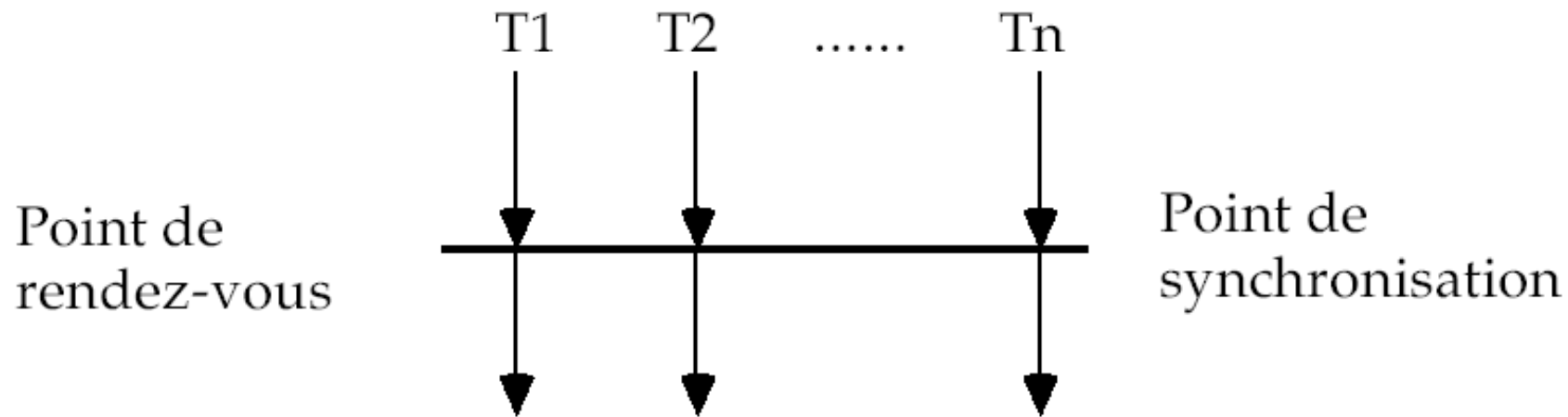


Synchronisation

Principe général : rendez-vous



Une tâche doit pouvoir :

- activer une autre tâche ;
- se bloquer (attendre) ou éventuellement bloquer une autre tâche ;
- débloquer une ou même plusieurs tâches.

Caractérisation des mécanismes de synchronisation

- déblocages mémorisés ou non ;
- mécanismes directs ou indirects.

Mécanismes de synchronisation

Mécanismes directs

`bloque(tâche)` *endort la tâche désignée*

`dort()` *endort la tâche qui l'exécute*

`réveille(tâche)` *réveille la tâche désignée*

Mécanismes indirects

- Synchronisation par objets communs, à travers des primitives protégeant les accès concurrents
- Synchronisation par sémaphores privés
 - 1 seule tâche peut exécuter `acquire` sur ce sémaphore ;
 - il est initialisé à 0 pour être toujours bloquant.

Synchronisation par événements

Un événement a deux états : arrivé ou non_arrivé

Manipulation par deux primitives :

- signaler (e) : indique que l'événement est arrivé
- attendre (e) : bloque la tâche jusqu'à ce que l'événement arrive

Ces deux primitives sont exécutées en EXCLUSION MUTUELLE

Synchronisation par événements

Événements mémorisés : association d'un booléen et d'une file d'attente

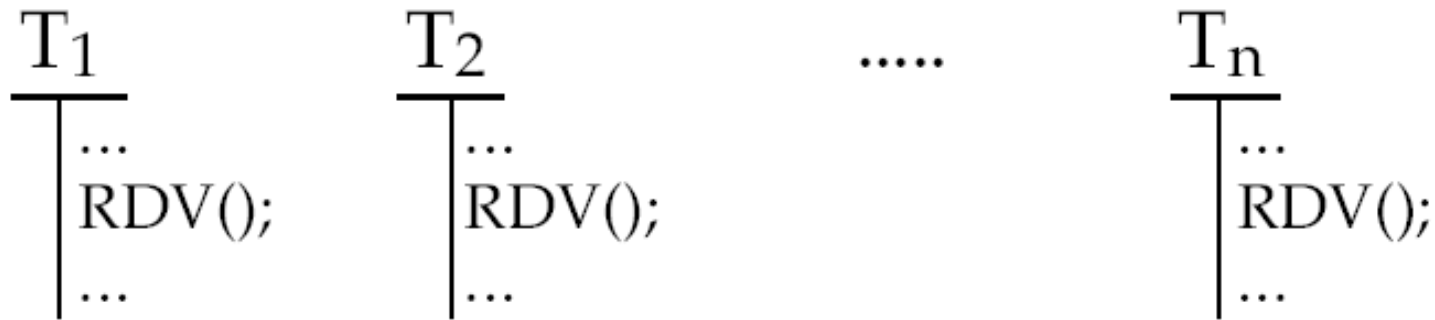
<pre>type Événement = structure arrivé : booléen; file : file d'attente; fin structure;</pre>	<pre>procédure signaler(modifié e:Événement) début e.arrivé ← vrai; <i>Réveiller toutes les tâches en attente</i> <i>dans e.file</i> fin signaler;</pre>
<pre>procédure r_a_z(modifié e:Événement) début e.arrivé ← faux; fin r_a_z;</pre>	<pre>procédure attendre(modifié e:Événement) début si non e.arrivé alors <i>Mettre la tâche dans e.file</i> dort(); finsi fin attendre;</pre>

Synchronisation par événements

Événements non mémorisés : simple file d'attente

type Événement = file d'attente;	procédure signaler(modifié e:Événement) début <i>Débloquer toutes les tâches en attente dans e</i> fin signaler;
Note : <i>attendre</i> est toujours bloquante	procédure attendre(modifié e:Événement) début <i>Mettre la tâche dans e.file</i> dort(); fin attendre;

Rendez-vous par événements



Comment écrire RDV ?

i : un entier;
 e : un Événement; {Mémemorisé}

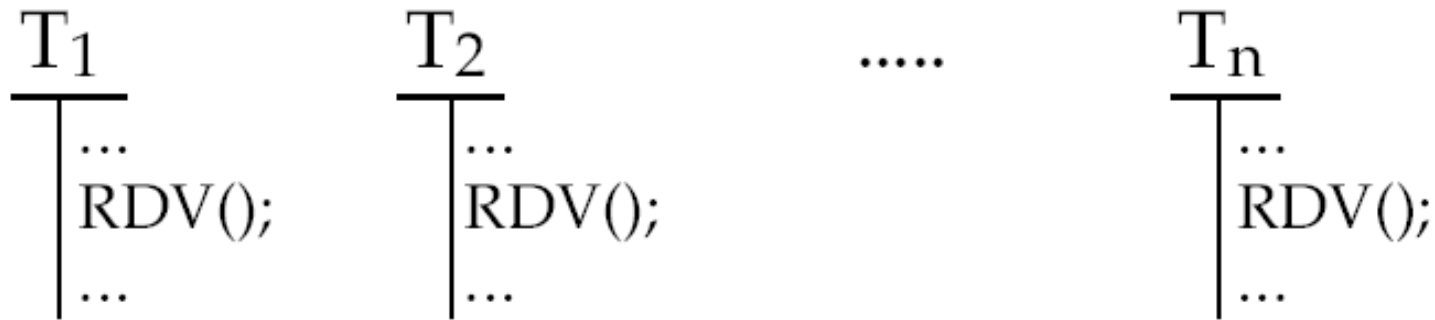
Initialisation :

$i \leftarrow 0$;
`r_a_z (e);`

```
procédure RDV();  
début  
     $i \leftarrow i + 1$ ;  
    si  $i < n$  alors  
        attendre(e)  
    sinon { $i = n$ }  
        signaler(e);  
    finsi  
fin RDV;
```

Pb : accès concurrent à i !

Rendez-vous par événements



Comment écrire RDV ?

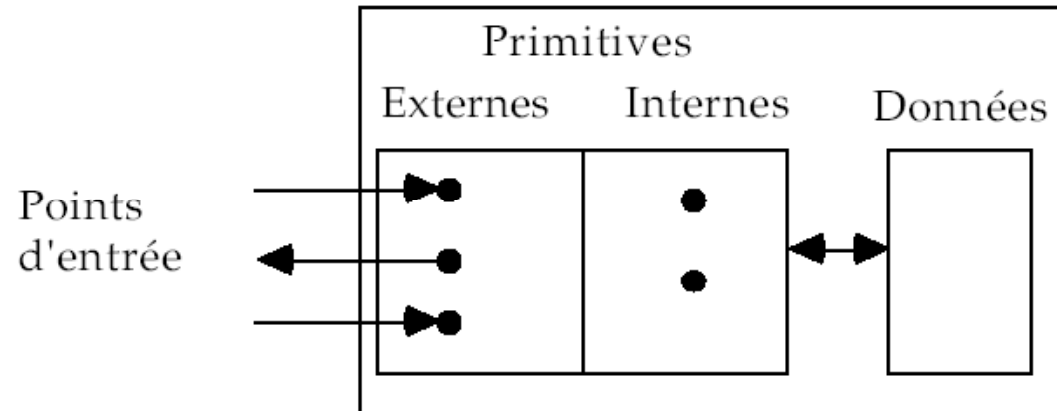
i : un entier;
 e : un Événement; {Mémoire}
 mutex : sémaphore

Initialisation :

$i \leftarrow 0$;
 $\text{r_a_z}(e)$;
 $\text{init_semaphore}(\text{mutex}, 1)$

```
procédure RDV();  
début  
    acquire(mutex)  
     $i \leftarrow i + 1$ ;  
    si  $i < n$  alors  
        release(mutex)  
        attendre( $e$ )  
    sinon { $i = n$ }  
        release(mutex)  
        signaler( $e$ );  
    finsi  
fin RDV;
```

Les moniteurs



Module avec des propriétés particulières

- on n'a accès qu'aux primitives externes (publiques), pas aux variables (privées) ;
- les procédures sont exécutées en exclusion mutuelle et donc les variables internes sont manipulées en exclusion mutuelle.
- on peut bloquer et réveiller des tâches. Le blocage et le réveil s'expriment au moyen de *conditions*.

Les moniteurs

Conditions = variables, manipulables avec :

procédure **attendre** (modifié c : condition);

début

 Bloque la tâche qui l'exécute et l'ajoute dans la file d'attente associée à c

Fin attendre;

fonction **vide** (c : condition) → un booléen;

début

 si la file d'attente associée à c est vide alors

 renvoyer(VRAI)

 sinon

 renvoyer(FAUX)

 finsi

Fin vide;

procédure **signaler** (modifié c : condition);

début

 si non vide(c) alors

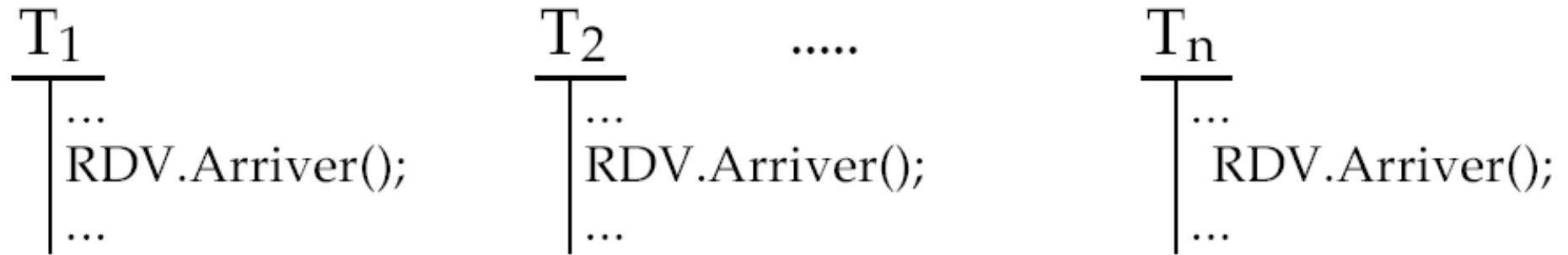
 extraire la première tâche de la file associée à c

 la réveiller

 finsi

Fin signaler;

Rendez-vous avec un moniteur



Moniteur RDV;

Point d'entrée

Arriver

Lexique

n : un entier constant = ??

i : un entier;

`tous_là` : une condition;

Début {Initialisations}

$i \leftarrow 0$;

Fin RDV;

procédure Arriver();

début

$i \leftarrow i + 1$;

si $i < n$ **alors**

attendre(`tous_là`);

finsi

signaler(`tous_là`); {Réveil en cascade}

fin Arriver;

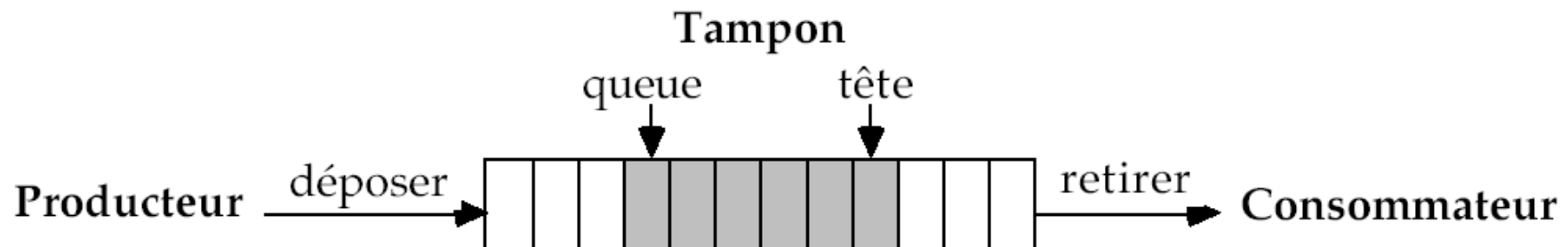
Communication entre tâches

Partage de mémoire commune + exclusion mutuelle

Modèle général : **producteur / consommateur**

Producteur → Données → **Consommateur**

- La communication est en général asynchrone : une des deux tâches travaille en général plus vite que l'autre.
- Pour limiter les effets de l'asynchronisme, on insère un tampon entre le producteur et le consommateur :



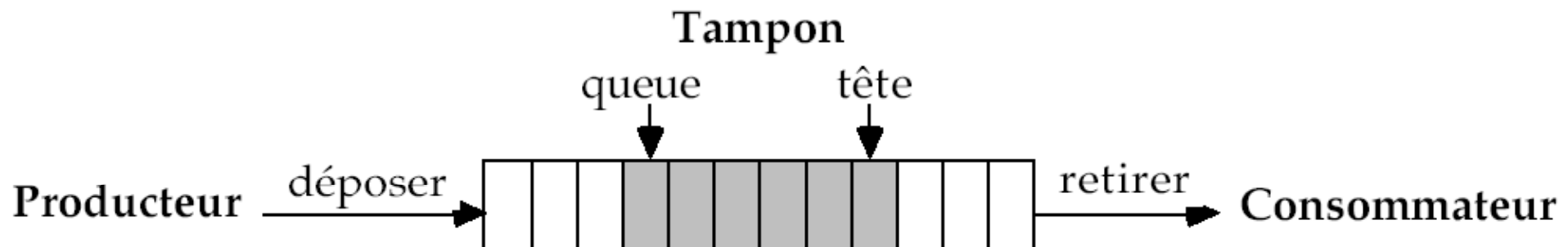
Producteur/Consommateur avec tampon

Producteur

```
tantque vrai faire  
    produire(m);  
    Tampon.déposer(m);  
fintantque
```

Consommateur

```
tantque vrai faire  
    Tampon.retirer(m);  
    consommer(m);  
fintantque
```



Producteur/Consommateur avec tampon

Moniteur Tampon; {Suppose Taille et Message connus}

Points d'entrée

déposer, retirer;

Lexique

nb : entier; {Nbre d'elt. dans le tampon (0..Taille)}

non_plein, non_vide : conditions; {Pour le blocage et le réveil}

tamp : tableau[0..Taille-1] de Messages;

tête, queue : entier; {0..Taille-1}

procédure **déposer** (consulté m:Message)

début

si nb = Taille alors

attendre(non_plein)

finsi

{Ajouter m au tampon}

nb $\leftarrow$ nb + 1;

tamp[queue] $\leftarrow$ m;

queue $\leftarrow$ (queue+1) mod Taille;

signaler(non_vide);

fin déposer;

procédure **retirer** (consulté m:Message)

début

si nb = 0 alors

attendre(non_vide)

finsi

{ Enlever m du tampon }

m $\leftarrow$ tamp[tête];

tête $\leftarrow$ (tête+1) mod Taille;

nb $\leftarrow$ nb - 1;

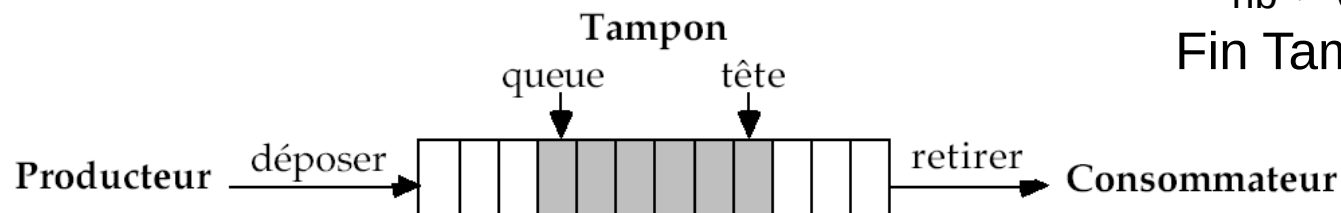
signaler(non_plein);

fin retirer;

Début

nb $\leftarrow$ 0; tête $\leftarrow$ 0; queue $\leftarrow$ 0;

Fin Tampon;



Démo en java...

Le problème des philosophes qui mangent et pensent...

- 5 philosophes qui mangent et pensent
- Pour manger il faut 2 fourchettes, droite et gauche
- On en a seulement 5 !
- Un problème classique de synchronisation
- Illustre la difficulté d'allouer ressources aux tâches tout en évitant interblocage et famine

