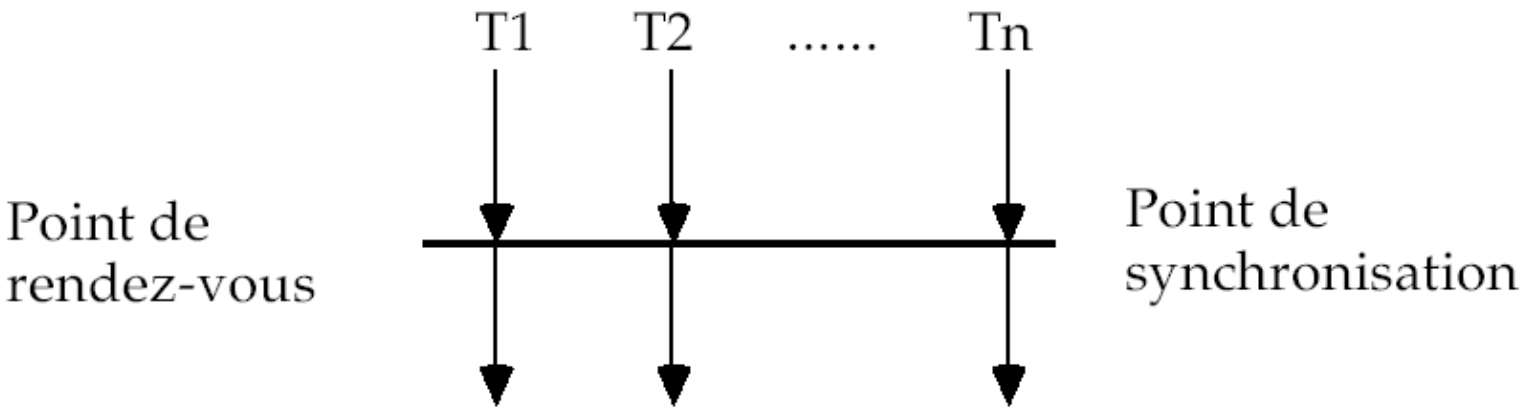
Synchronisation

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

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

```
Classe Evénement
    privé arrivé : booléen;
    privé file : file d'attente;

public procédure r_a_z()
début
     arrivé ← faux;
fin r_a_z;

fin Evénement
```

```
public procédure signaler()
début
     arrivé ← vrai;
     Réveiller toutes les tâches en attente
     dans file
fin signaler;

public procédure attendre()
début
     si non arrivé alors
          Mettre la tâche dans file
          dormir();
     finsi
fin attendre;
```

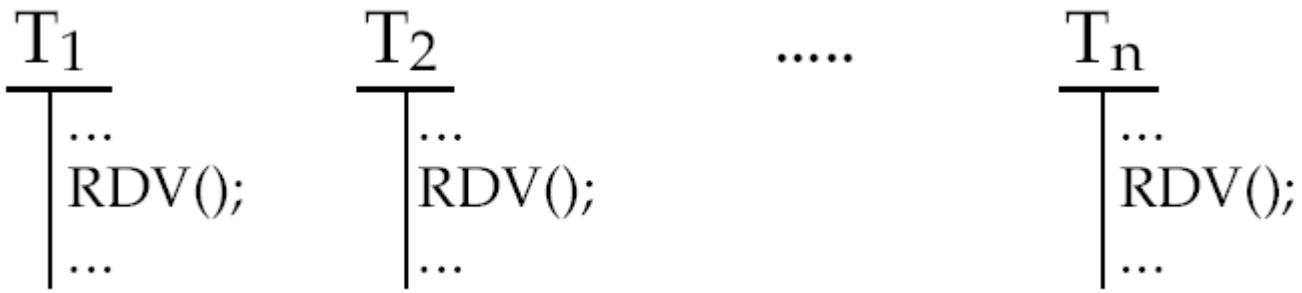
Synchronisation par événements

Evénements non mémorisés : simple file d'attente

Classe Evénement_nonmémorisé **privé** file : file d'attente;	**public procédure** signaler() **début** *Débloquer toutes les tâches en attente* *dans e* **fin** signaler;
	public procédure attendre() **début** *Mettre la tâche dans file* dormir(); **fin** attendre;

Note : *attendre* est toujours bloquante

Rendez-vous par événements



Comment écrire RDV ?

```
i : entier;
e : Evénement; {Mémorisé}
```

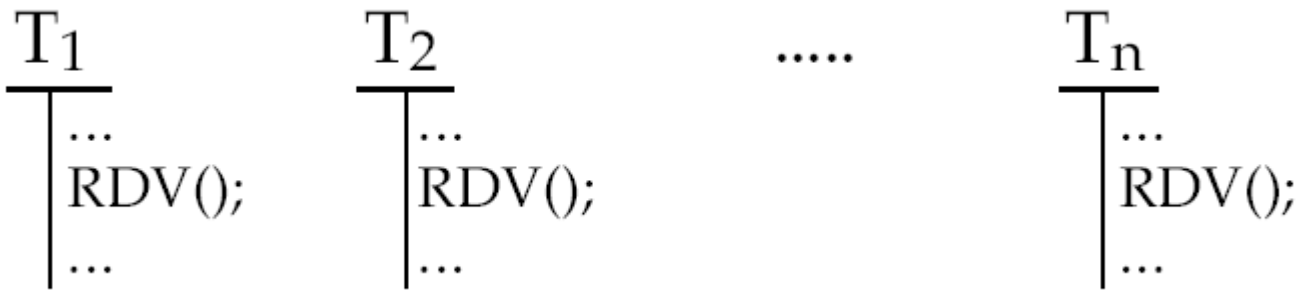
Initialisation :

```
i ← 0;
e.r_a_z ();
```

Pb : accès concurrent à i !

```
procédure RDV();
début
    i ← i + 1;
    si i < n alors
            e.attendre()
    sinon   // i = n
            e.signaler();
    finsi
fin RDV;
```

Rendez-vous par événements



Comment écrire RDV ?

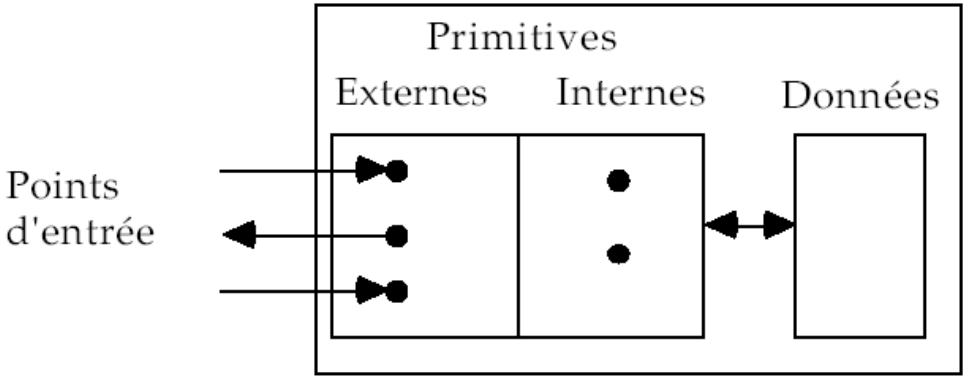
```
i : un entier;
e : un Evénement; {Mémorisé}
mutex : Sémaphore
```

Initialisation :

```
i ← 0;
e.r_a_z ();
mutex.init_semaphore(1)
```

```
procédure RDV();
début
    mutex.aquire()
    i ← i + 1;
    si i < n alors
        mutex.release()
        e.attendre()
    sinon   // i = n
        mutex.release()
        e.signaler();
    finsi
fin RDV;
```

Les Moniteurs



Classe avec des propriétés particulières

- on n'a accès qu'aux primitives publiques, les variables sont privées
- les **procédures sont exécutées en exclusion mutuelle** et donc les variables privées 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

Classe Condition

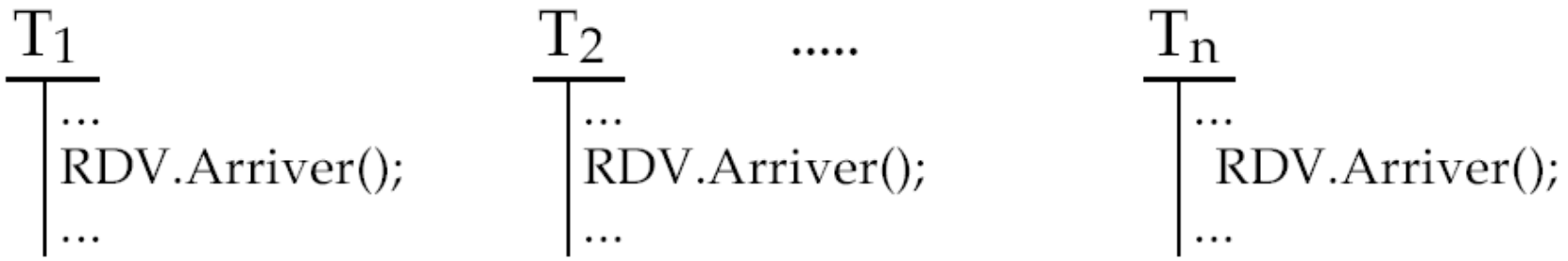
privé file : file d'attente

public procédure attendre ();
début
    Bloque la tâche qui l'exécute et l'ajoute dans la file d'attente
fin attendre;

public fonction vide () → un booléen;
début
    si file est vide alors
        renvoyer(VRAI)
    sinon
        renvoyer(FAUX)
    finsi
fin vide;

public procédure signaler ();
début
    si non vide() alors
        extraire la première tâche de la file d'attente
        la réveiller
    finsi
fin signaler;

Rendez-vous avec un moniteur



Classe RDV

n : entier

i : entier

tous_là : Condition

public RDV(nbtâches) //constructeur
début

i ← 0

n ← nbtâches

fin

fin RDV;

public procédure Arriver()
début

i ← i + 1

si i < n **alors**

tous_là.attendre()

finsi

tous_là.signaler() // Réveil en cascade

fin Arriver;

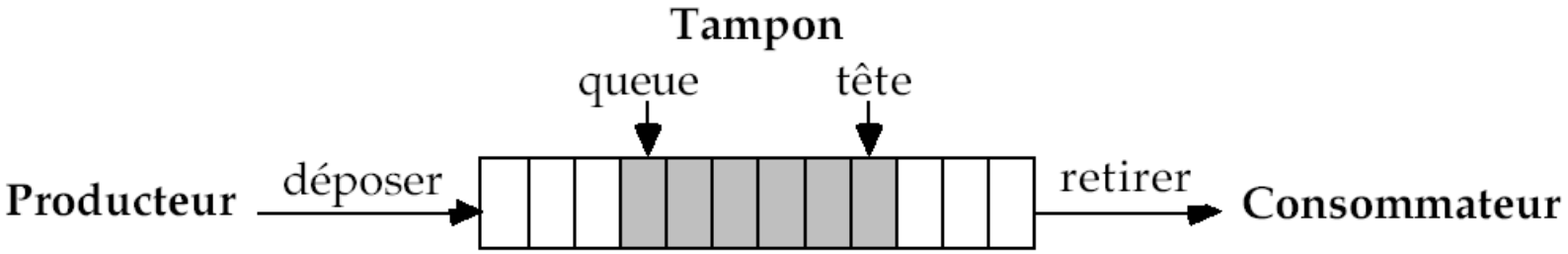
Communication entre tâches

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 :



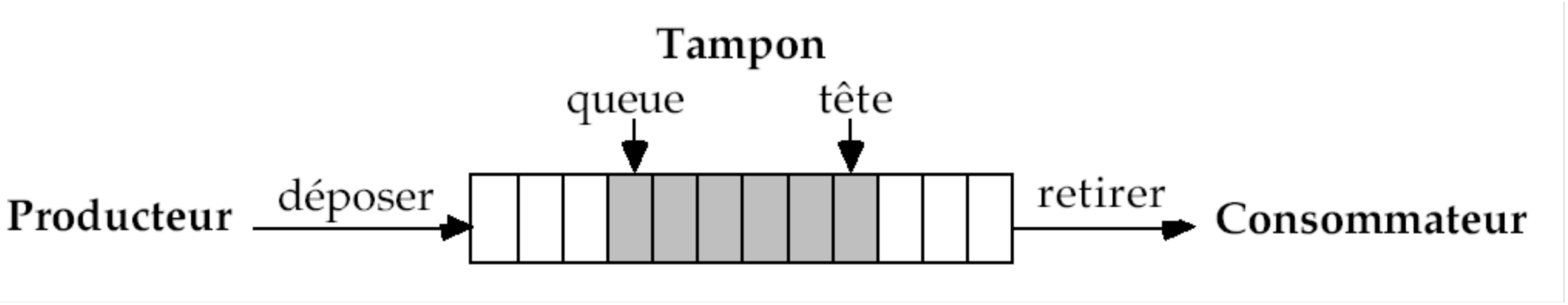
Producteur/Consommateur avec tampon

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'élt. 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 sur 0..Taille-1
```

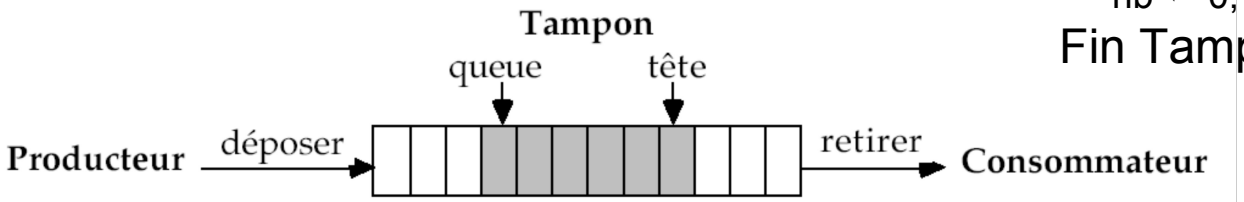
```
procédure déposer (consulté m : Message)
début
    si nb = Taille alors
        attendre(non_plein)
    finsi
    {Ajouter m au tampon}
    nb ← nb + 1;
    tamp[queue] ← m;
    queue ← (queue+1) mod Taille;
    signaler(non_vide);
fin déposer;
```

```
procédure retirer (consulté m : Message)
début
    si nb = 0 alors
        attendre(non_vide)
    finsi
    { Enlever m du tampon }
    m ← tamp[tête];
    tête ← (tête+1) mod Taille;
    nb ← nb - 1;
    signaler(non_plein);
fin retirer;
```

Début

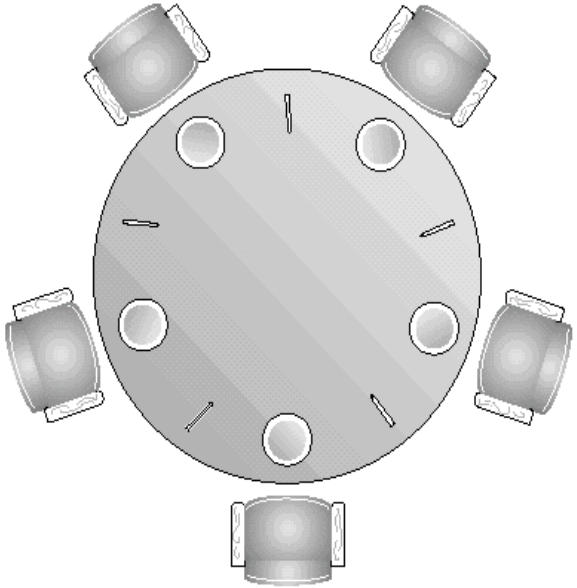
```
nb ← 0; tête ← 0; queue ← 0;
```

Fin Tampon;



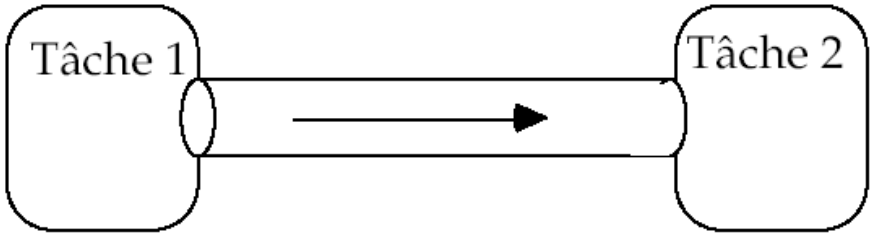
Le problème des philosophes qui mangent et pensent…

- 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



Communication par tubes

Principe : flot de communication entre deux tâches



Modèle à tampon potentiellement infini (l'écriture n'est jamais bloquante) avec plusieurs messages à la fois

Exemple : les tubes Unix (pipes)

Un tube Unix est l'association de deux flots de communication, l'un utilisé en écriture, l'autre utilisé en lecture.

Création

```
int tube[2];
pipe(tube);
```

Lecture, bloquante si le tube est vide

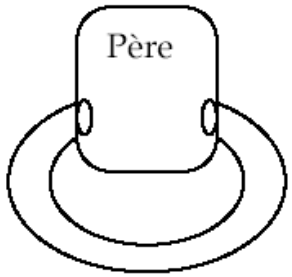
```
int x, nombre;
char chaine[MAX];
x = read(tube[0], chaine, nombre);
```

Ecriture, rarement bloquant

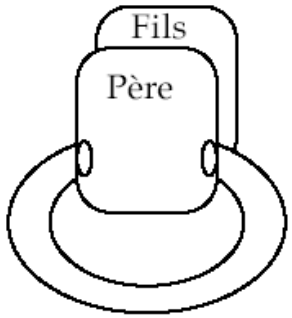
```
int x, nombre;
char chaine[MAX];
x = write(tube[1], chaine, nombre);
```

Exemple en langage C

```
void main(void)
{
     int tube[2];
     char message_recu[8];
     /* Création du tube, précède le fork */
     pipe(tube);
```




```
     if ((ident = fork()) == -1)
          perror(fork);
     else
```




```
     if (ident == 0) {
          close(tube[0]);
          write(tube[1],
                "Bonjour",8);
          }
     else {
          close(tube[1]);
          read(tube[0],
             message_recu,8);
     }
}
```