INF2610
Noyau d’un système d’exploitation

Chapitre 5 - Moniteurs

Sommaire

- Qu'est qu'un moniteur ?
- Comment assurer l'exclusion mutuelle ?
- Limite des moniteurs
- Variables de condition d'un moniteur
- Problème des producteurs et des consommateurs
- Comment implémenter les moniteurs et les variables de condition à l'aide de sémaphores ?
- Exercices

Qu’est ce qu’un moniteur ?

- C’est une structure de données composée d’attributs et de méthodes qui partagent en exclusion mutuelle ces attributs.

- C’est le compilateur du langage qui assure cette exclusion mutuelle en ajoutant, lors de la compilation, le code nécessaire.

- Si un processus P1 est actif dans le moniteur (c-à-d en exécution d’une méthode du moniteur) sur un objet et qu’un autre processus P2 demande l’accès au moniteur (c-à-d invoque une méthode du moniteur) sur ce même objet alors P2 est mis en attente du moniteur jusqu’à ce que P1 libère l’accès.
  ➔ **Chaque moniteur a une file d’attente gérée FIFO**.

- Les moniteurs sont supportés par les langages de programmation C# et Java.

Comment assurer l'exclusion mutuelle ?

- Pour assurer l'exclusion mutuelle, il suffit de regrouper toutes les sections critiques et toutes les variables partagées d'un problème dans un même moniteur.

- **Exemple de moniteur** : Compte d'une banque

```
Moniteur Compte
{  int solde  = 0 ;

   void Deposer (int  montant) // section critique pour le dépôt
   {  solde = solde + montant  ;
    }

   void retirer (int montant)    //section critique pour le retrait
   {  if (solde >= montant)
            solde = solde – montant ;
   }
}
```

Cette solution est plus simple que les sémaphores, car le programmeur n'a pas à se soucier de l'exclusion mutuelle. Elle est assurée par le moniteur.

Limite des moniteurs

- Les moniteurs à eux seuls ne permettent pas d'implémenter certains problèmes de synchronisation comme, par exemple, le problème des producteurs et des consommateurs.

- Considérez le moniteur ProducteursConsommateurs (voir le transparent suivant) composé :

  - des variables partagées : tampon, ic, ip , compteur (qui indique le nombre d'objets dans le tampon) et

  - des méthodes (sections critiques) deposer et retirer appelées respectivement par les producteurs et les consommateurs.

Limite des moniteurs

```
Moniteur ProducteursConsommateurs
{   const int N= 100;
     int tampon[N];
     int compteur =0, ic=0, ip=0 ;
     void deposer (int objet) // section critique pour le dépôt
     {          while (compteur==N);
                tampon[ip] = objet ;
                ip = (ip+1)%N ;
                compteur++ ;
      }
     void retirer (int& objet) //section critique pour le retrait
     {          while (compteur ==0) ;
                objet = tampon[ic] ;
                ic = (ic+1)%N ;
                compteur -- ;
     }
}
```

Le consommateur va rentrer dans une attente active infinie dans le moniteur, s'il accède en premier au moniteur (car le tampon est vide). Le producteur va se retrouver en attente infinie du moniteur.

Limite des moniteurs

- Risques d'interblocage :
  1. Le processus actif dans le moniteur est un consommateur alors que le tampon est vide. Le consommateur rentre dans une attente active jusqu'à ce que le tampon devienne non vide. Les producteurs ainsi que les autres consommateurs ne pourront plus accéder au moniteur.

  2. Le processus actif dans le moniteur est un producteur alors que le tampon est plein. Le producteur rentre dans une attente active jusqu'à que le tampon devienne non plein. Les autres producteurs ainsi que les consommateurs ne pourront plus accéder au moniteur.

- Pour pallier cette limitation, il suffit d'offrir la possibilité de se mettre en attente passive dans le moniteur tout en libérant l'accès au moniteur ➔ **Variables de condition**.

Variables de condition

- Une variable de condition est une variable qui représente une condition / un événement. Elle n'a pas de valeur explicite mais a une file d'attente (gérée FIFO).

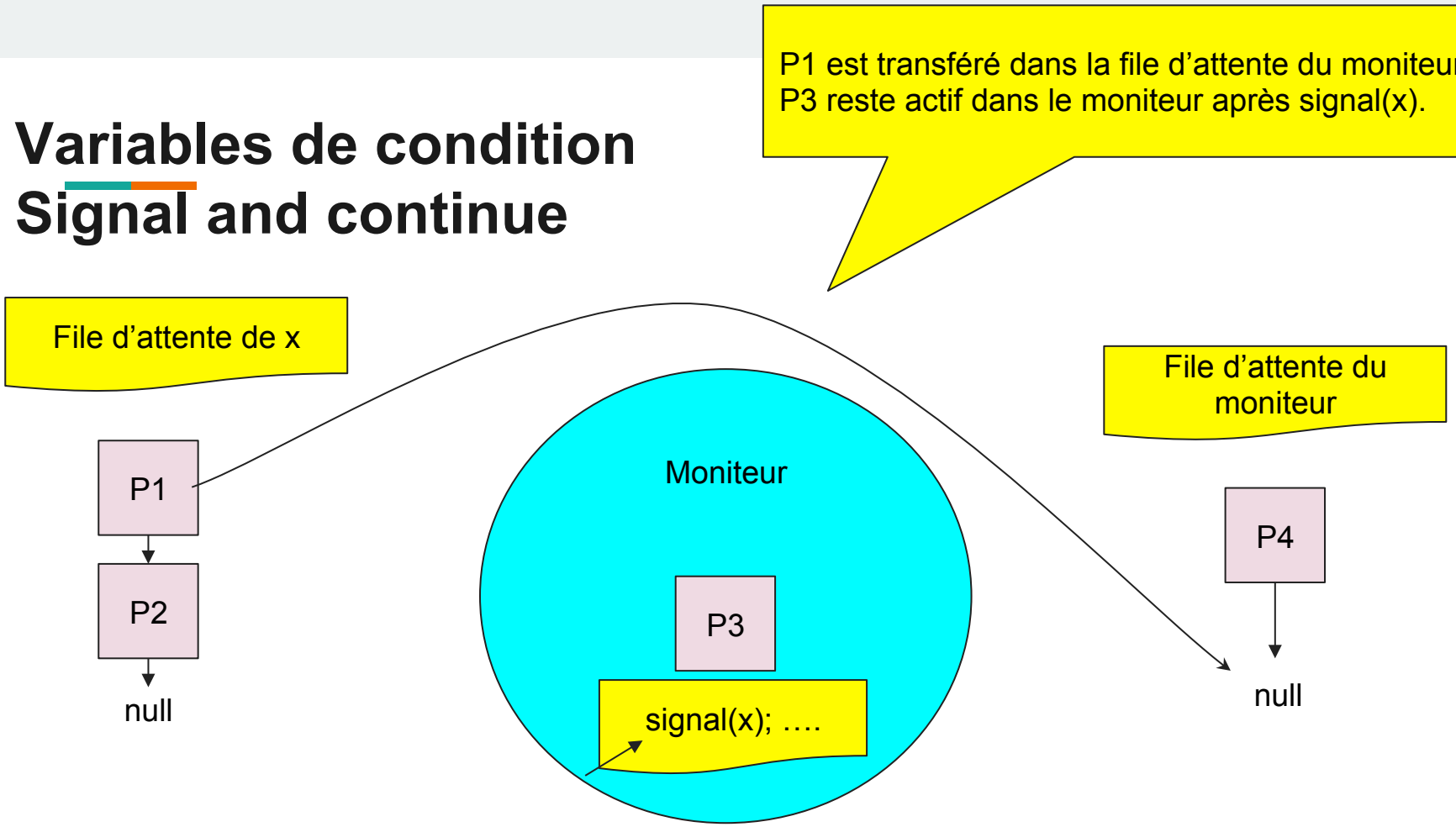
- Une variable de condition x est **manipulée par deux opérations de base :**
  - wait(x) : suspend l'exécution du processus/thread appelant pour le mettre en attente de x (dans la file d'attente de x) et
  - signal(x) : débloque un processus/thread qui est en attente de x. Le processus/thread débloqué est mis :
    - > **en attente du moniteur (dans la fille d'attente du moniteur)** **ou**
    - > **en attente du processeur (dans la file d'attente des processus/threads prêts)**. Dans ce dernier cas, l'exécution du processus/thread appelant est suspendu. Il est mis en attente dans une file d'attente supplémentaire appelée **file d'attente des processus/threads suspendus dans le moniteur**.

    ➔ Deux sémantiques différentes pour signal(x).

Signal and continue

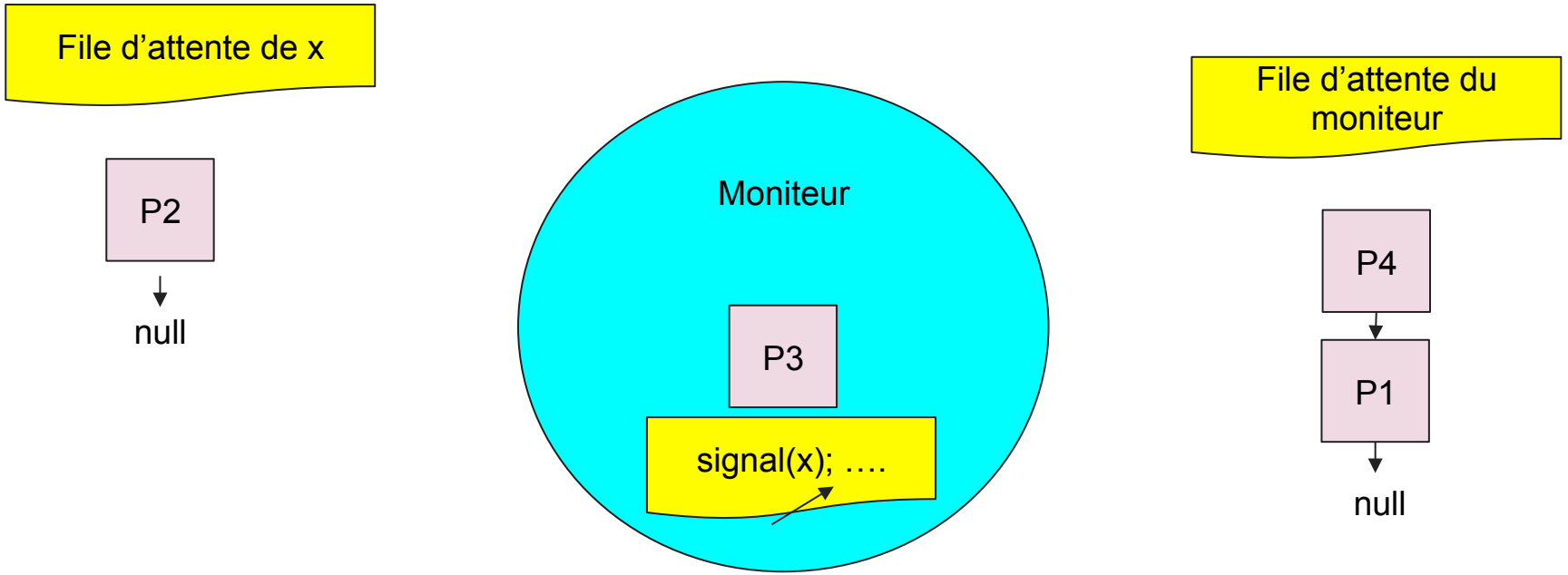
Signal and wait

Variables de condition
Signal and continue



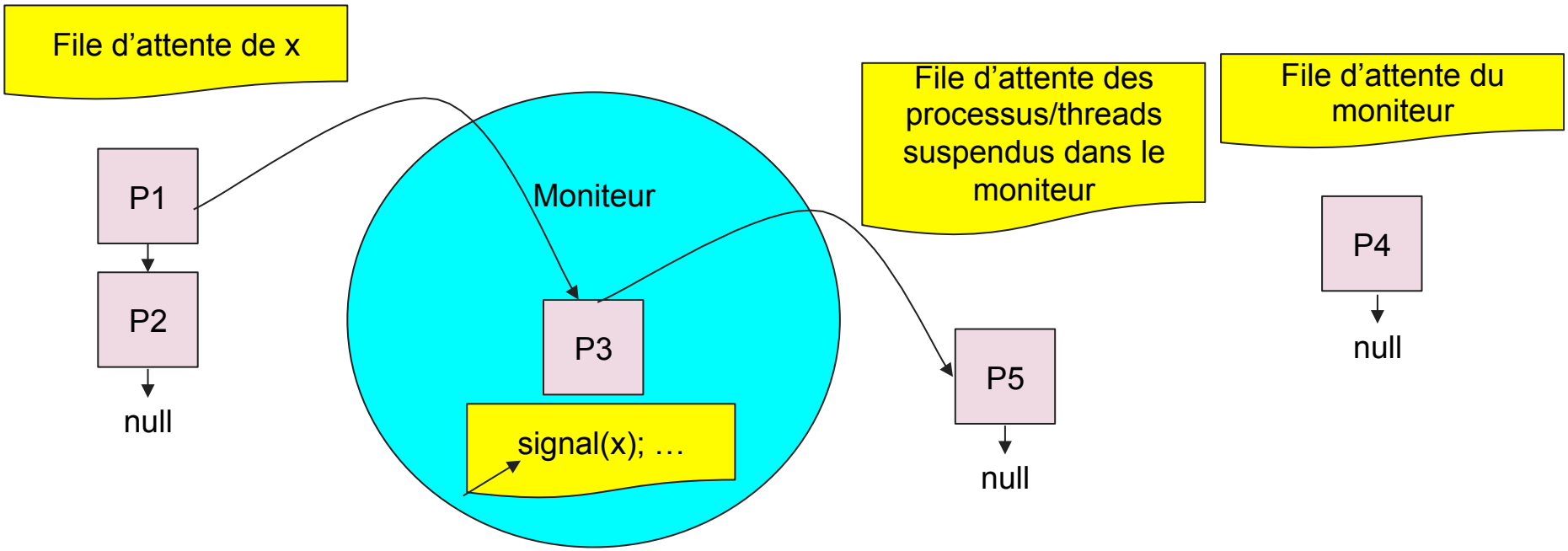
Variables de condition
Signal and continue

P1 est transféré dans la file d'attente du moniteur.
P3 reste actif dans le moniteur après signal(x).

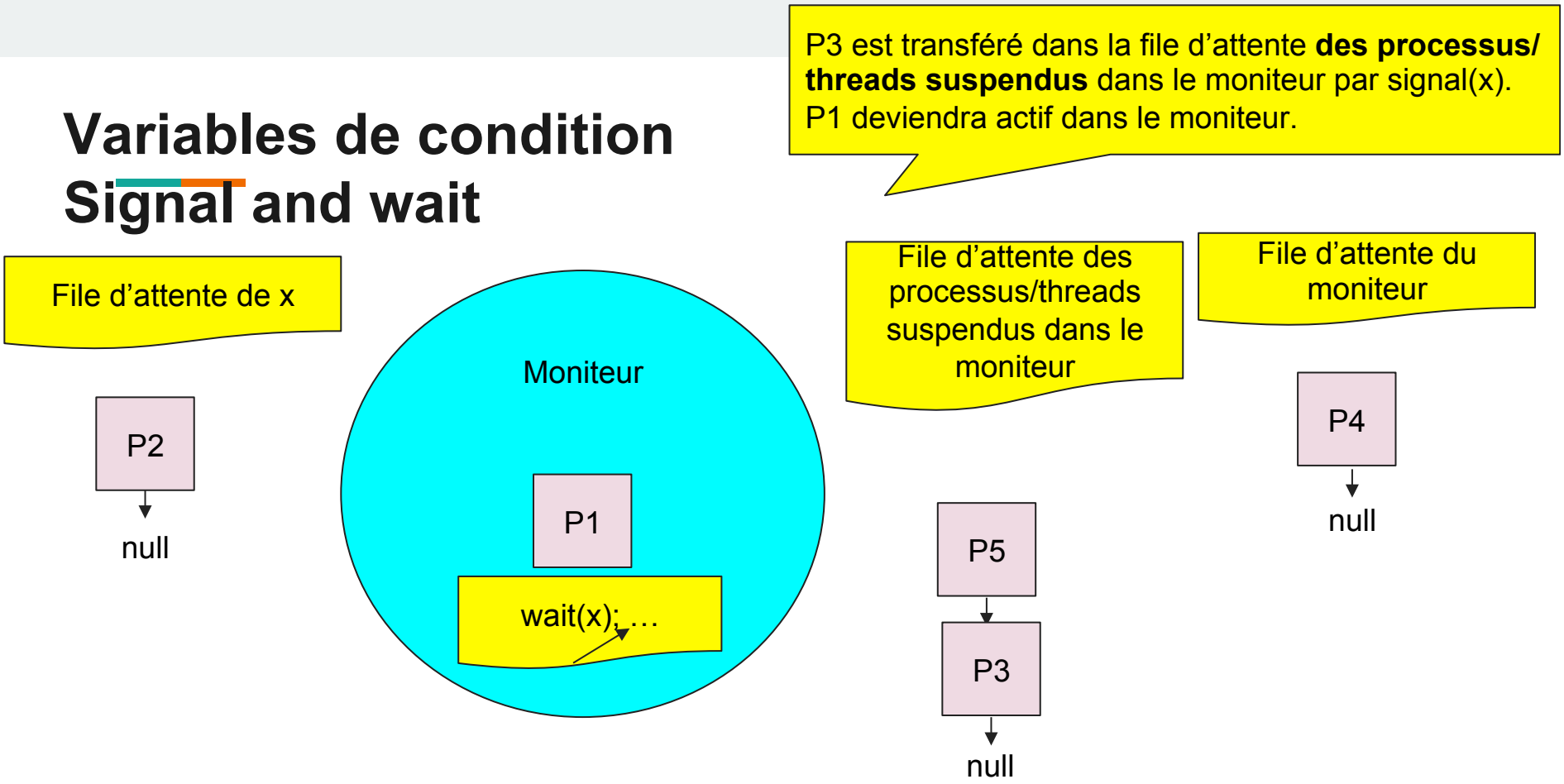


Variables de condition
Signal and wait

P3 est transféré dans la file d'attente **des processus/ threads suspendus** dans le moniteur par signal(x).
P1 deviendra actif dans le moniteur.



Variables de condition
Signal and wait



- Dans ce cours, nous considérons la sémantique « signal and continue » comme dans les langages JAVA et C#.

Problème des producteurs et des consommateurs

```
Moniteur ProducteurConsommateur
{   const int N= 100;
     int tampon[N];        boolc  nplein, nvide;
     int compteur =0, ic=0, ip=0 ;
     void deposer (int objet) // section critique pour le dépôt
     {              while (compteur==N);    while(compteur==N) wait(nplein);
                    tampon[ip] = objet ;
                    ip = (ip+1)%N ;
                    compteur++ ; if(compteur==1) signal(nvide);
      }
     void retirer (int& objet) //section critique pour le retrait
     {              while (compteur ==0) ;  while(compteur==0) wait(nvide);
                    objet = tampon[ic] ;
                    ic = (ic+1)%N ;
                    compteur -- ; if(compteur==N-1) signal(nplein);
     }
}
```

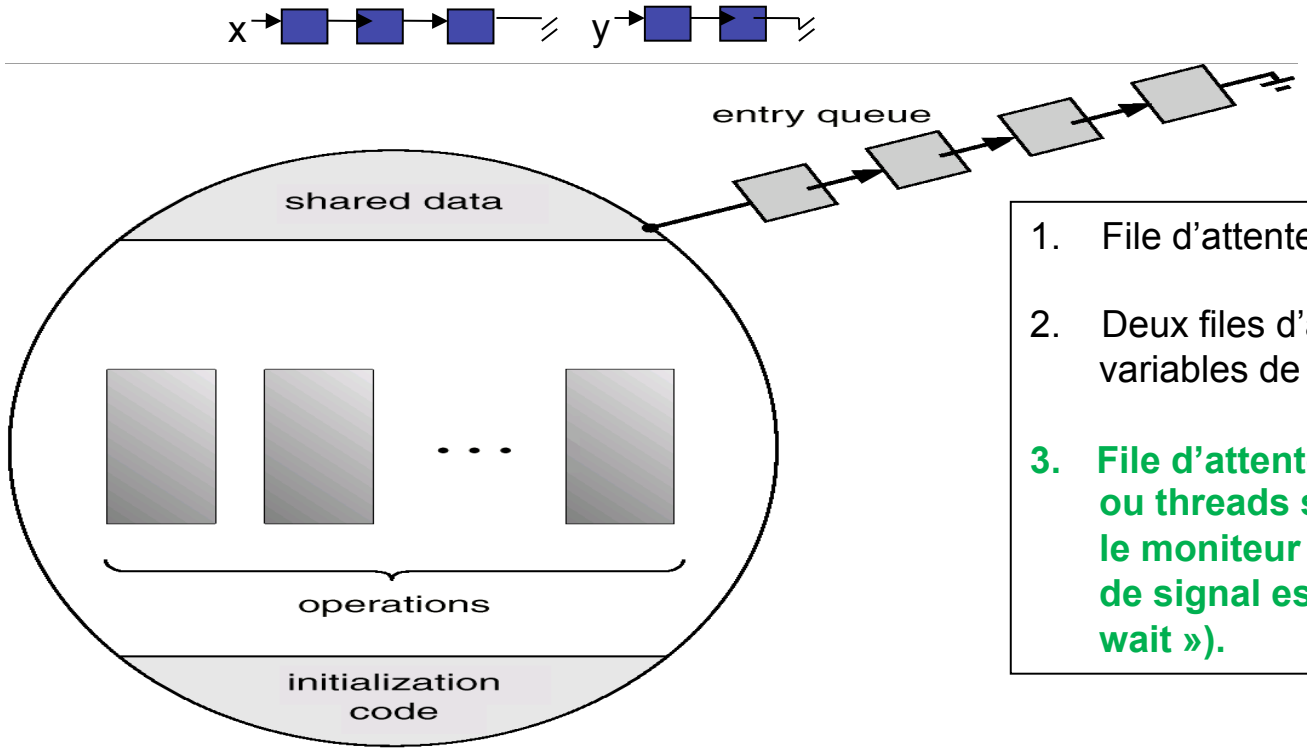
boolc est un type utilisé ici pour désigner les variables de condition

Comment implémenter les moniteurs et les variables de condition à l'aide de sémaphores ?

L'implémentation nécessite :

- une file d'attente pour mémoriser les demandes d'accès au moniteur (file d'attente du moniteur),
- une file d'attente pour chaque variable de condition (wait(x)), et
- éventuellement, une file d'attente des processus suspendus dans le moniteur suite à l'opération signal(x) (dans le cas où la sémantique est **« signal and wait »**).

Comment implémenter les moniteurs et les variables de condition à l'aide de sémaphores ? (2)



1. File d'attente du moniteur.
2. Deux files d'attente pour les variables de condition x et y.
3. **File d'attente des processus ou threads suspendus dans le moniteur (si la sémantique de signal est « signal and wait »).**

Comment implémenter les moniteurs et les variables de condition à l'aide de sémaphores ? (3)

Sémantique « signal-and-continue » :

- La **file d'attente d'accès au moniteu**r est simulée par celle d'un sémaphore binaire **mutex; Ce sémaphore sert à assurer les accès en exclusion mutuelle aux méthodes du moniteur.**

- La **file d'attente de chaque variable de condition** est simulée par celle d'un **sémaphore dont la valeur est toujours 0.** Il faut aussi un compteur du nombre de processus/threads dans la file d'attente. Elle sert à maintenir la valeur du sémaphore à 0.

Comment implémenter les moniteurs et les variables de condition à l'aide de sémaphores ? (3)

Sémantique « signal-and-continue » :

Avant

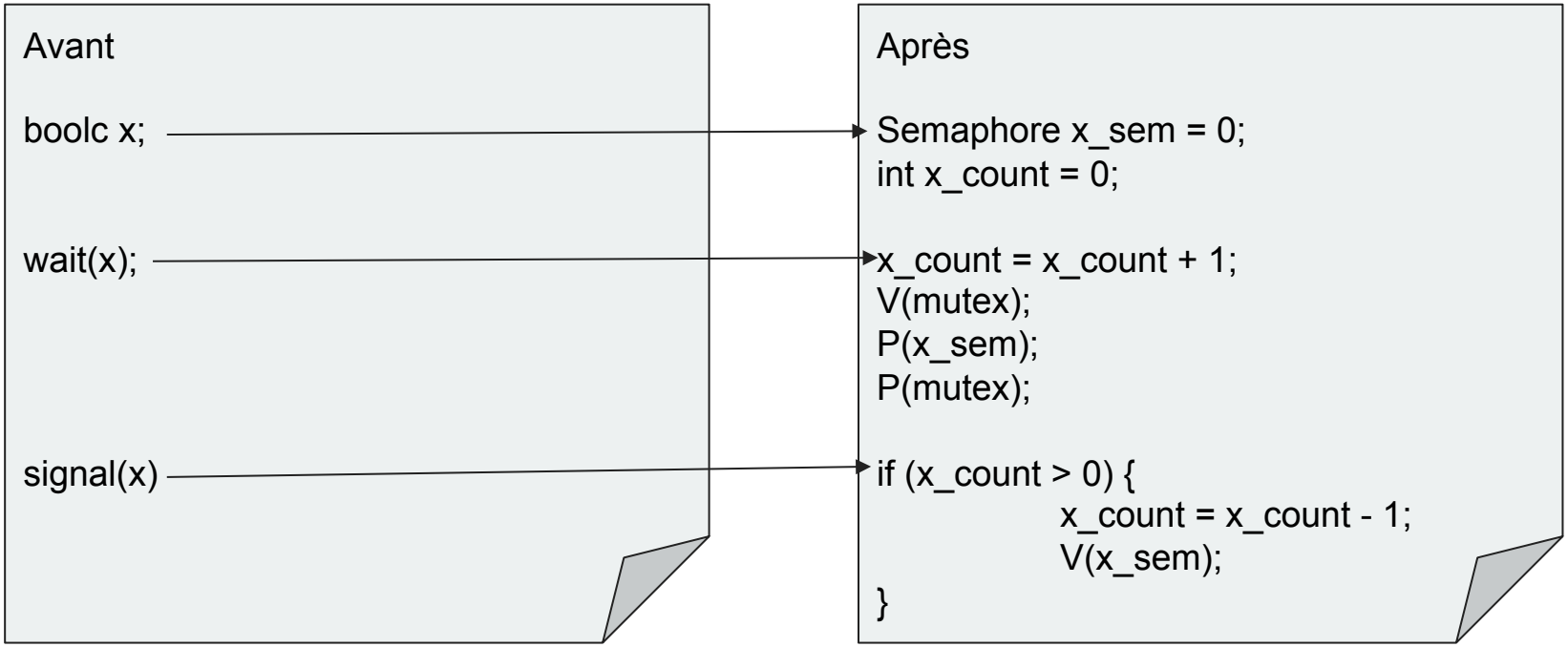
```
// une méthode du moniteur
Type  F ( type1 param1, …, typen paramn)
{

    // corps de la fonction



     return …;
}
```

Après

```
Semaphore mutex = 1; // partagé par toutes les méthodes

Type  F ( type1 param1, …, typen paramn)
{
     P(mutex);

     // corps de la fonction


      V(mutex);
     return …;
}
```

Comment implémenter les moniteurs et les variables de condition à l'aide de sémaphores ?

Sémantique « signal-and-continue » :



Exercice 1

- Implémentez un sémaphore (opérations P et V) à l'aide du Moniteur suivant. Vous devez compléter les fonctions P et V.
- Vous avez le droit de déclarer des variables de condition et des variables standards (int, etc.) ainsi que d'utiliser les opérations wait et signal des variables de condition.

```
Moniteur Semaphore (int v0) {

    void P() {

    }

    void V() {

    }

}
```

Exercice 2

Pour permettre à un ensemble de threads d'un même processus de partager des données de type pile, on décide d'implémenter un moniteur *pile_t*. On vous donne le code à compléter du moniteur *pile_t* :

```
Moniteur pile_t
{     const  int  N = 2;  // la  taille de la pile
      int  P[N];   // la pile
      ….  // autres variables ou constantes
      void  Empiler ( int o)  { …. }; // doit bloquer si la pile est pleine
      int  Depiler ( )  { … .};// doit bloquer si la pile est vide
}
```

Complétez le moniteur pile_t.

Moniteurs JAVA

- Un moniteur est associé à tout objet JAVA.
- Les méthodes de l'objet qui doivent accéder à l'objet en exclusion mutuelle sont déclarées de type « synchronized ».
- Une seule variable de condition «implicite » et les méthodes wait, notify (l'équivalent de signal) et notifyAll sont associées à tout objet JAVA.
- Les moniteurs JAVA sont de type « signal and continue ».
- Chaque objet JAVA a deux files d'attente :
  - Entry queue : file d'attente du moniteur
  - Wait queue : file d'attente de la variable de condition.