Moniteurs (1/7)

- **Un autre mécanisme de synchronisation**
  - De plus haut niveau que les sémaphores
- **Intégré à certains langages de programmation**
  - Exemple : Java
- **Contexte**
  - Un moniteur est associé à un type de données abstrait
  - Un type de données abstrait définit :
    - Des variables privées
    - Des méthodes/procédures publiques pour manipuler les variables (ainsi qu'une méthode d'initialisation)
  - Un moniteur est une forme particulière de type de données abstrait qui garantit que les méthodes sont exécutées en exclusion mutuelle
    - On a donc, au plus, un seul processus actif au sein d'un moniteur

Moniteurs (2/7)

- **Variables de Conditions**
  - Aussi appelées « conditions » ou « condition variables »
  - Mécanisme complémentaire de l'exclusion mutuelle entre méthodes
  - Une variable de condition est associée à une file d'attente
  - Permet de définir des schémas de synchronisation applicatifs (conditions de blocage/déblocage de processus)
  - On peut définir une ou plusieurs variables de conditions au sein d'un moniteur

Moniteurs (3/7)

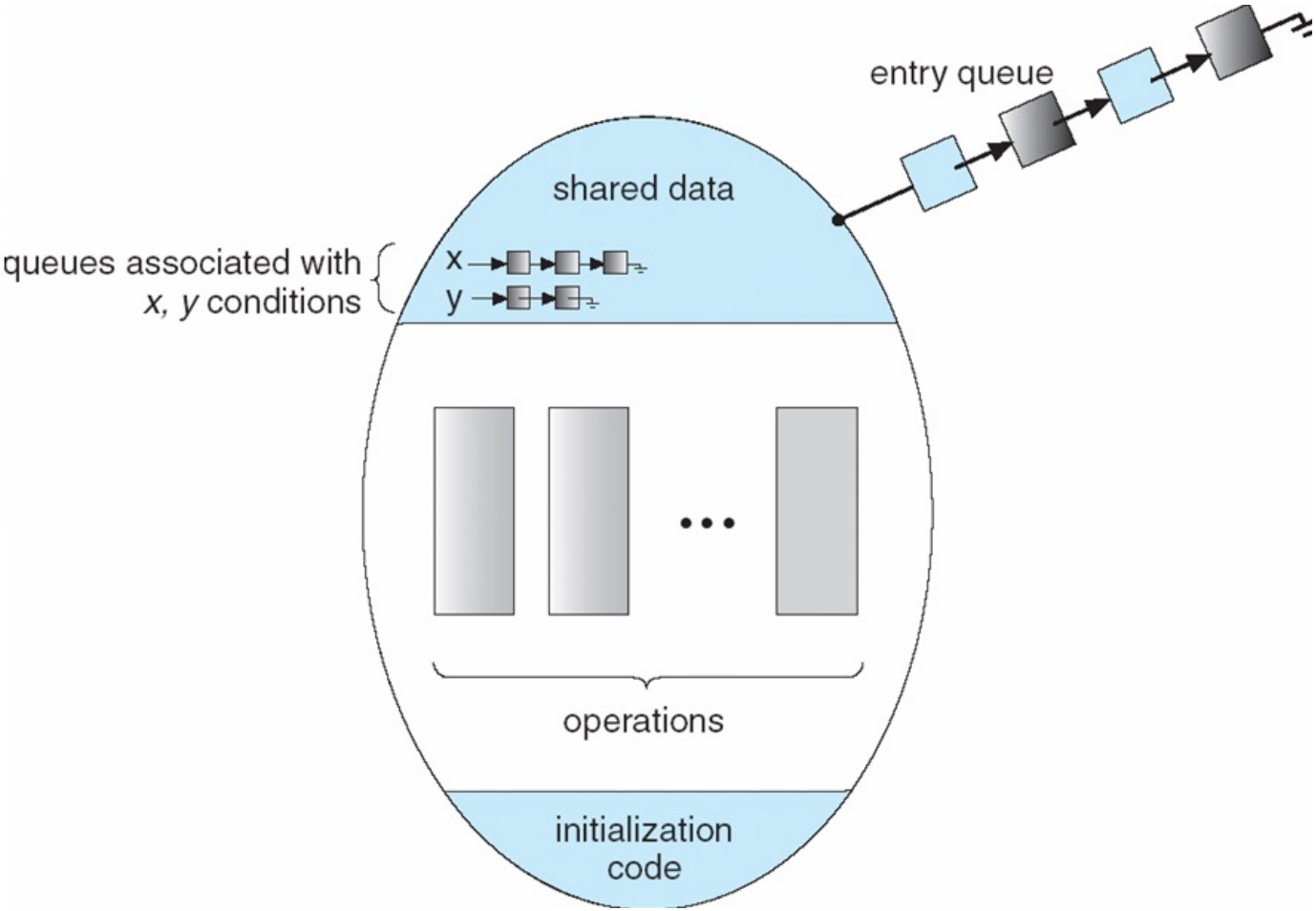
- **Variables de Conditions (suite)**
  - Une variable de condition fournit les opérations suivantes :
    - **`wait`** : bloque le processus appelant sur la variable de condition correspondante et libère le moniteur
    - **`signal`** (ou **`notify`**) : réveille l'un des processus bloqués sur la variable de condition correspondante
    - **`broadcast`** : réveille tous les processus bloqués sur la variable de condition correspondante

Moniteurs (4/7)

- **Avertissements concernant les variables de conditions**
  - Attention : **réveils intempestifs (« spurious wakeups »)**
    - La plupart des implémentations de la primitive **`wait`** peuvent être sujettes à des réveils arbitraires (c'est-à-dire non déclenchés par un appel à **`signal`** ou **`broadcast`**). Ce genre de situation (bien qu'assez rare en pratique) doit impérativement être pris en compte par les programmeurs qui utilisent **`wait`** afin de garantir une exécution correcte.
  - Attention : **Ordre de réveil variable selon les spécifications**
    - Pas forcément de garantie FIFO
  - Attention : **Signaux fugaces**
    - Un signal de réveil est « perdu » si aucun processus bloqué au moment de l'appel à **`signal`**
    - Contrairement aux sémaphores (un appel à V incrémente toujours le compteur)

Moniteurs (5/7)

- **Vue schématique**



Source : Silberschatz et al. Operating systems concepts.

Moniteurs (6/7)

- **Détails concernant les primitives `wait/signal`**
  - Lorsqu'un processus P1 invoque **`signal`** sur une variable de condition, un processus P2 qui était bloqué sur cette variable est débloqué
  - Mais, par définition, P1 et P2 ne peuvent se retrouver actifs en même temps dans le moniteur (propriété d'exclusion mutuelle)
  - Différentes sémantiques possibles (*"signaling disciplines"*) :
    - ***Signal and wait*** (priorité au processus signalé)
      - P1 est suspendu jusqu'à ce que P2 quitte le moniteur (fin de la méthode appelée par P2) ou que P2 se bloque sur un appel à **`wait`**
    - ***Signal and continue*** (priorité au processus signalant)
      - P2 est suspendu jusqu'à ce que P1 quitte le moniteur (fin de la méthode appelée par P1) ou que P1 se bloque sur un appel à **`wait`**

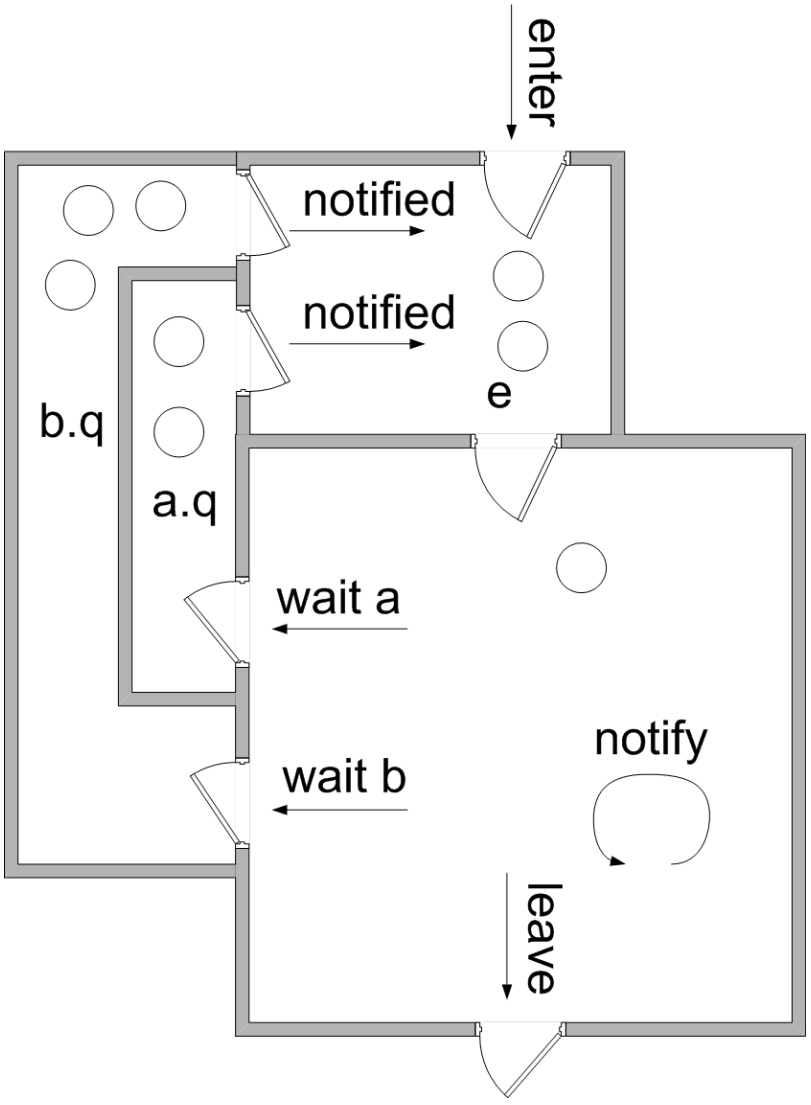
Moniteurs (7/7) - Synthèse des principales règles

- **Au maximum un seul processus actif dans le moniteur**
- **Lors d'un appel à `signal`, selon les spécifications :**
  - Soit le processus signalant garde le moniteur
  - Soit le processus signalé prend le moniteur
- **Libération du moniteur par un processus**
  - Lorsque l'exécution de la procédure synchronisée est terminée
  - Lors d'un appel à **`wait`**
  - Pour certaines spécifications, lors d'un appel à **`signal`** (cf. point précédent)
- **Lors de la libération d'un moniteur, selon les spécifications, le moniteur peut être attribué :**
  - Soit en priorité à un processus suspendu au sein du moniteur (un processus signalant ou signalé, selon la politique choisie lors d'un appel à **`signal`**)
  - Soit à n'importe quel processus qui souhaite obtenir le moniteur
- **Il est important de connaître les spécifications précises du langage/système considéré pour programmer correctement**

Moniteurs - synthèse

Une autre vue schématique,

pour la sémantique *« signal and continue » :*



Cas d'étude : Parking N places avec moniteurs

```
monitor parking {
    int N;
    cond_t c;   // condition
    ...

    void init() {
        N = ...; // nombre de places max
        cond_init(&c);
    }

    void entrer_parking() {    // methode synchronisee
        while(N==0)   // noter le "while" (et non pas "if")
            wait(&c); // attendre
        N--;
    }

    void sortir_parking() {    // methode synchronisee
        N++;
        signal(&c);
    }

}
```

Tests des conditions (1/2)

- **Faire très attention aux blocs de code qui contiennent un appel à `wait`**
  - Rappel: lorsque la sémantique d'un moniteur est différente de *"signal and wait"*, un appel à la primitive **`signal`** ne déclenche pas un basculement immédiat/atomique vers le processus débloqué
  - Conséquence : l'état de la condition booléenne testée par un processus P1 peut évoluer entre le moment auquel un processus P2 débloque P1 et le moment auquel P1 prend la main
  - Conclusion : dans la plupart des cas (mais pas tous), un appel à **`wait`** doit être encadré dans une boucle **`while`** (pour revérifier la condition de déblocage) et non pas dans une clause **`if`**
- **Exemple**
  - Reprenons le cas du parking N places
    - En remplaçant **`while(N==0)`** par **`if(N==0)`**

Tests des conditions (2/2)

- **Exemple (suite) : séquence d'exécution possible**
  - Etat de départ : N vaut 0

P1 (entrer_parking)	P2 (sortir_parking)	P3 (entrer_parking)
`if (N==0)`		
`wait(&c);`		
	`N++;`	
	`signal(&c);`	
		`if (N==0)`
		`N--;`
`N--;`		

Primitive `broadcast`

- **(concerne la sémantique *"signal and continue"*)**
- **Un appel à `broadcast` ne réveille que les processus bloqués sur la variable de condition concernée**
- **Attention**
  - Un appel à **`broadcast`** est souvent inefficace car, pour la plupart des processus réveillés, la condition qu'ils attendent ne sera pas/plus satisfaite lorsqu'ils arriveront à obtenir le verrou d'exclusion mutuelle. Ils devront donc se bloquer à nouveau.
  - **Cependant, attention aux tentatives d'optimisations naïves en remplaçant systématiquement un appel à `broadcast` par un appel à `signal`.** Une telle optimisation n'est correcte que si toutes les hypothèses suivantes sont réunies:
    - On ne cherche pas à réveiller un processus particulier (ou bien on est certain que seul ce processus est actuellement bloqué sur la variable de condition)
    - Tous les processus bloqués sur la variable de condition sont en attente de la même condition logique
    - Seul un processus à la fois (parmi l'ensemble des processus bloqués sur la variable de condition) peut bénéficier d'un réveil (ou bien on gère le réveil des multiples processus en cascade : le premier réveille le second, etc.)

Moniteurs Java

- **Un moniteur est associé à chaque objet et à chaque classe**
- **Chaque moniteur englobe un verrou et une (seule) variable de condition**
  - Le verrou est manipulé implicitement (cf. mot clé **`synchronized`**)
  - La variable de condition est manipulée via les méthodes **`wait`**, **`notify`** et **`notifyAll`**
- **Mot clé `synchronized` pour définir les méthodes synchronisées (au niveau de la signature d'une méthode)**
  - Toutes les méthodes ne sont pas nécessairement synchronisées
- **On peut aussi utiliser les moniteurs Java à l'échelle d'un bloc de code au sein d'une méthode**
  - Bloc **`synchronized`** – On doit préciser en paramètre sur quel objet on se synchronise, c'est-à-dire l'objet dont on utilise le verrou et la variable de condition

Sections critiques conditionnelles

- **Autre mécanisme de synchronisation**
- **Assez proche des moniteurs**
  - Similarité : utilisation de variables de conditions
  - Différence : gestion manuelle de l'exclusion mutuelle à l'aide de verrous
- **Exemple : primitives `pthread_cond_wait/signal/...` pour les threads POSIX (programmation C)**
- **Interface de programmation**
  - Primitives de verrouillage déjà vues :
    - **`void mutex_lock(mutex_t *m),`** ...
  - Primitives de gestion des conditions :
    - **`void cond_init(cond_t *c)`**
    - **`void cond_wait(cond_t *c, mutex_t *m)`**
      - Libère le verrou et bloque sur la condition
      - Lors du déblocage, reprend le verrou
      - Comme pour les moniteurs, la plupart des implémentations peuvent être sujettes à des réveils interpestifs
    - **`void cond_signal(cond_t *c)`**
    - **`void cond_broadcast(cond_t *c)`**

Cas d'étude : Parking N places avec sections critiques conditionnelles

```
int N = ... // nombre de places max
mutex_t m;  // à initialiser avec mutex_init
cond_t c;   // à initialiser avec cond_init
...

void entrer_parking() {
    mutex_lock(&m);
    while(N==0)
        cond_wait(&c, &m); // attendre
    N--;
    mutex_unlock(&m);
}

void sortir_parking() {
    mutex_lock(&m);
    N++;
    cond_signal(&c, &m);
    mutex_unlock(&m);
}
```

Sections critiques conditionnelles – Discussion (1/2)

- **La libération du verrou doit-elle obligatoirement être gérée au sein de l'appel à `cond_wait` ?**
  - Oui
  - **`cond_wait(cond_t *c, mutex_t *m)`** libère le verrou **`m`** et bloque le processus sur la condition **`c`** de manière atomique
  - L'atomicité est nécessaire. Sinon, risque d'entrelacement problématique

- **Exemple**
  - **Remplaçons `cond_wait(&c, &m)` par la séquence : `mutex_unlock(&m); cond_wait(&c); mutex_lock(&m);`**
  - **Considérons deux processus :**
    - **P1 exécute `entrer_parking()`**
    - **P2 exécute `sortir_parking()`**
    - **Initialement, `N` vaut 0 (aucune place disponible)**

Sections critiques conditionnelles – Discussion (2/2)

- **Séquence d'exécution possible :**

P1 (entrer_parking)	P2 (sortir_parking)
`...`	
`mutex_unlock(&m);`	
	`mutex_lock(&m);`
	`N++;`
	`cond_signal(&c);`
	`mutex_unlock(&m);`
`cond_wait(&c);`	
`...`	

Remarque

- **Nous avons étudié différents mécanismes de synchronisation**
  - Mutex
  - Sémaphores
  - Moniteurs
  - Section critiques conditionnelles

- **Ces mécanismes sont équivalents**
  - Autrement dit, on peut implémenter chacun d'entre eux en s'appuyant sur l'un des autres mécanismes