Principes des systèmes d'exploitation

Synchronisation

UFR IM^{2}AG

M1 MEEF NSI 2022-2023

Renaud Lachaize

Crédits et remerciements

- **Le contenu de ce support de cours est partiellement inspiré, voire emprunté aux travaux d'autres personnes :**

  - Rodrigue Chakode, Fabienne Boyer, Vincent Danjean, Sacha Krakowiak, Arnaud Legrand, Vania Marangozova-Martin (UFR IM^{2}AG)
  - David Mazières (Stanford University)
  - Randall Bryant, David O'Hallaron, Gregory Kesden, Markus Püschel (Carnegie Mellon University)

  - Ouvrages de référence (voir détails sur la page web) :
    - Silberschatz et al. Operating systems concepts with Java.
    - Tanenbaum. Modern operating systems.
    - Bryant and O'Hallaron. Computer systems: a programmer's perspective.

Objectifs du cours

- **Comprendre les besoins de synchronisation entre différents flots d'exécution (processus ou threads)**

- **Comprendre les mécanismes qui permettent de réaliser la synchronisation**
  - Implémentation
  - Utilisation correcte et efficace

Introduction

- **Flots d'exécution concurrents**
  - Threads au sein d'un même processus
  - Différents processus
  - **Par la suite, sauf mention contraire, on utilisera le terme « processus » de manière générique**

- **Situation de compétition ou de coopération entre processus**

  - **Compétition** : plusieurs processus veulent accéder à la même ressource (par exemple, modifier le même fichier ou la même variable)

  - **Coopération** : plusieurs processus interagissent pour mener à bien une tâche – ils doivent communiquer régulièrement pour échanger des informations et déterminer l'avancement global

  - Les deux situations nécessitent de synchroniser les processus

Exemple introductif n°1 : Compte bancaire

- **Deux opérations sur le même compte exécutées en concurrence**

processus p1 : créditer(1867A, 1000)

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 1000
3. ecrire_compte (1867A, nouveau)
```

processus p2 : créditer(1867A, 3000)

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 3000
3. ecrire_compte (1867A, nouveau)
```

- **À noter :**
  - les variables **`courant`** et **`nouveau`** sont locales à chaque processus
  - les deux processus se déroulent en parallèle. L'exécution des opérations peut être entrelacée dans un ordre quelconque, à condition de respecter l'ordre local pour chacun des processus
- **Exemple de séquences d'exécution**
  - Exécution n°1 : p2.1 ; p2.2 ; p2.3 ; p1.1 ; p1.2 ; p1.3
  - Exécution n°2 : p1.1 ; p1.2 ; p2.1 ; p2.2 ; p2.3 ; p1.3

Exemple introductif n°2 : tampon producteur/consommateur (1/2)

- Producteur

```
while (true) {

  // produce an item in
  // nextProduced

  while (count == BUFFER_SIZE)
    ; // do nothing

  buffer[in] = nextProduced;
  in = (in + 1)%BUFFER_SIZE;
  count++;
}
```

- Consommateur

```
while (true) {

  while (count == 0)
    ; // do nothing

  nextConsumed = buffer[out];
  out = (out + 1)%BUFFER_SIZE;
  count--;

  // consume the item in
  // nextConsumed
}
```

Exemple introductif n°2 : tampon producteur/consommateur (2/2)

- En langage machine

count++ se traduit par :

```
register1 = count
register1 = register1 + 1
count = register1
```

count-- se traduit par :

```
register2 = count
register2 = register2 - 1
count = register2
```

- Considérons la séquence suivante, avec initialement la valeur 5 stockée dans count

S0: prod. exécute register1 = count {register1 = 5}
S1: prod. exécute register1 = register1 + 1 {register1 = 6}
S2: cons. exécute register2 = count {register2 = 5}
S3: cons. exécute register2 = register2 - 1 {register2 = 4}
S4: prod. exécute count = register1 {count = 6 }
S5: cons. exécute count = register2 {count = 4}

- Conclusion ?

Le problème de l'exclusion mutuelle

Comment éviter les problèmes d'incohérences introduits par des accès concurrents à une ressource partagée ?

Assurer que l'ensemble des opérations (consultation + mise à jour) est exécutée de manière **indivisible** (« atomique »)

Ainsi, pas d'interférences possibles de la part d'autres opérations exécutées en parallèle

Exemple :

A1

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 1000
3. ecrire_compte (1867A, nouveau)
```

A2

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 3000
3. ecrire_compte (1867A, nouveau)
```

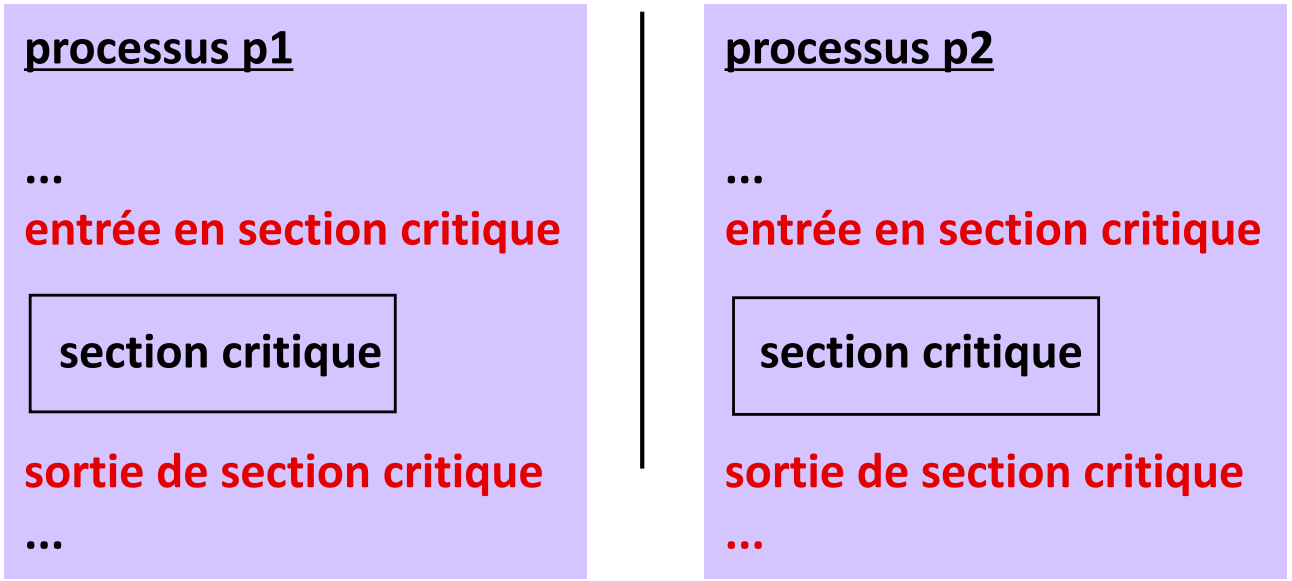
Si A1 et A2 sont atomiques, le résultat de l'exécution parallèle de A1 et A2 ne peut être que celui de A1 ; A2 ou de A2 ; A1, à l'exclusion de tout autre.

On dit aussi que la séquence d'actions 1; 2; 3 (dans A1 et A2) est une **section critique** : elle doit être exécutée en **exclusion mutuelle** (un seul processus au plus peut être dans sa section critique à un instant donné)

Section critique

- **Schéma général**

déclaration et initialisation de variables communes



Les opérations “entrée en section critique”, ”sortie de section critique” doivent garantir l’exclusion mutuelle

Attention : Ne faire aucune hypothèse sur les vitesses d’exécution relatives des différents processus

Section critique

- **Trois propriétés requises pour une solution au problème de la section critique**

- **Exclusion mutuelle**
  - Si un processus Pi se trouve dans la section critique, aucun autre processus ne peut y entrer.
- **Progrès**
  - Si aucun processus ne se trouve dans la section critique et un ou plusieurs processus souhaitent y entrer, alors seuls les processus candidats à la section critique peuvent participer au choix du prochain processus admis en section critique, et ce choix est fait au bout d'un temps borné.
- **Attente bornée**
  - Quand un processus demande à entrer en section critique, il existe une limite sur le nombre d'admissions d'autres processus dans cette section critique qui ont lieu entre le moment où la demande est faite et le moment où la demande est acceptée.

Réalisation d'une section critique

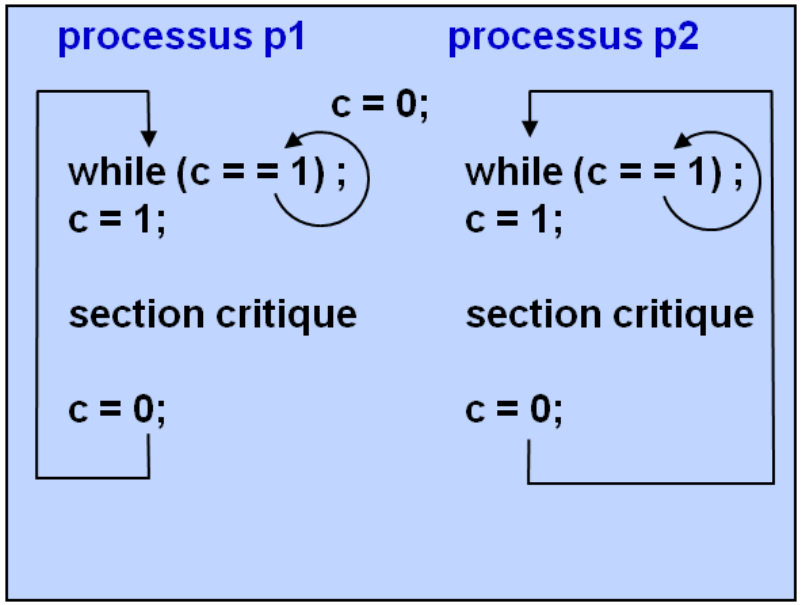
- **Pour commencer, considérons une approche aussi simple que possible, dite par « attente active »**
  - Un processus qui veut entrer en section critique boucle jusqu'à ce qu'il soit admis
  - Approche **inefficace** (en particulier dans un système monoprocesseur) – à éviter

- **Autres hypothèses :**
  - Seulement deux processus
  - Système monoprocesseur

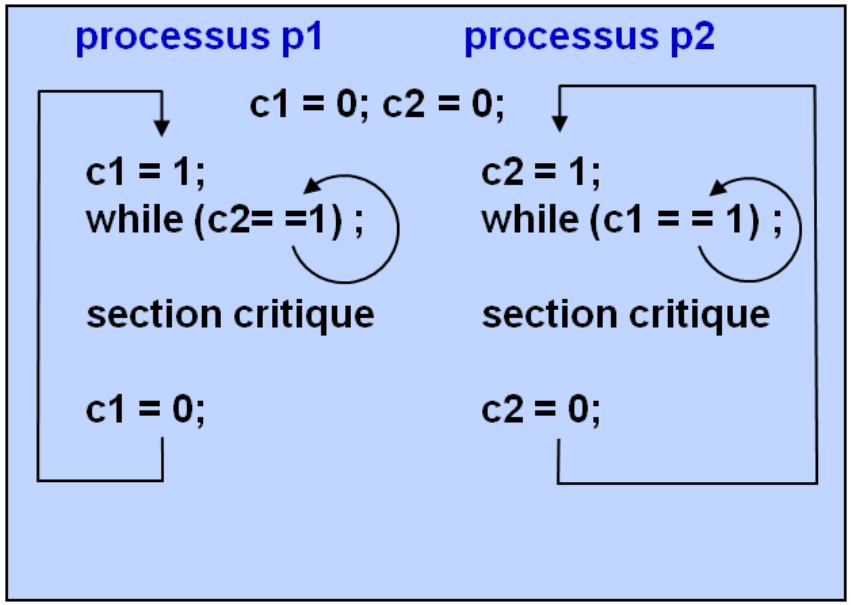
- **Même dans un contexte aussi simple, le problème de la section critique n'est pas trivial ...**

Réalisation d'une section critique par attente active

Proposition n°1

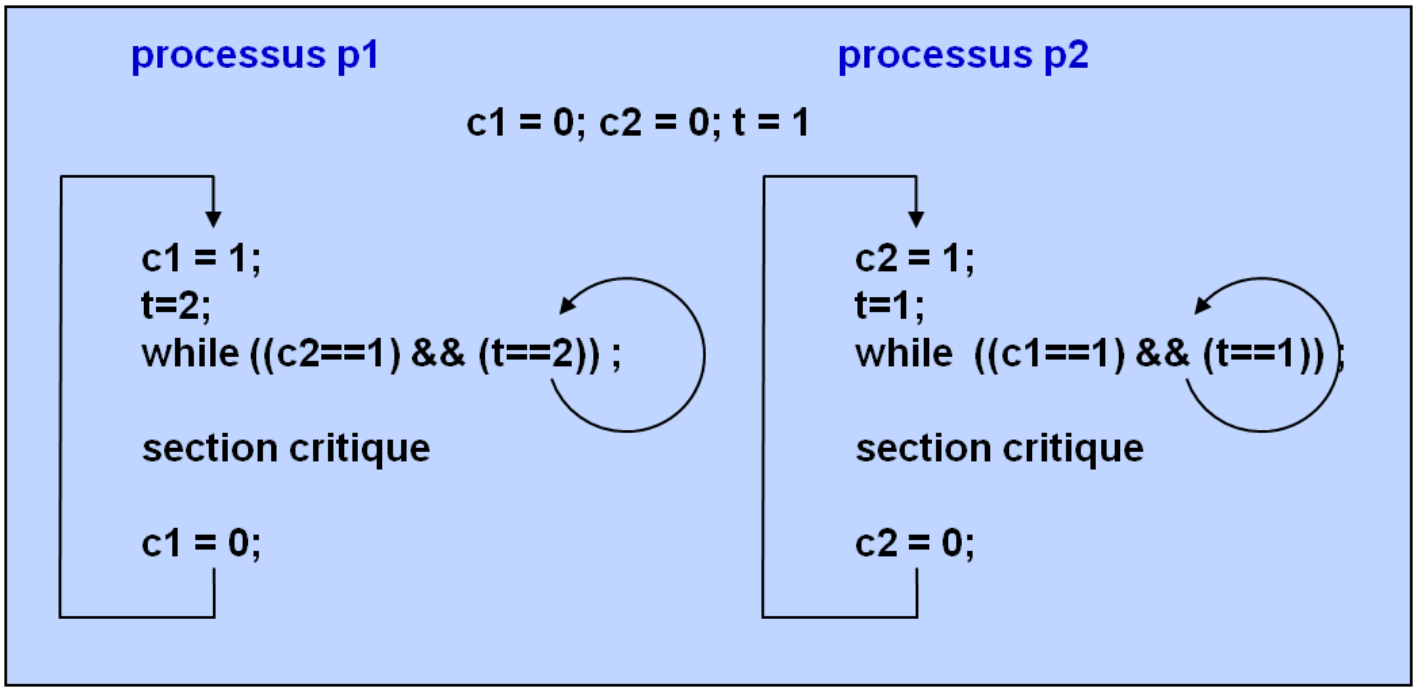


Proposition n°2



Réalisation d'une section critique par attente active (suite)

Une solution correcte pour l'exclusion mutuelle par attente active avec 2 processus (Peterson, 1981)



Exercice : prouver que cette solution respecte les 3 propriétés liées au problème de la section critique

Réalisation d'une section critique
Solution plus générale

- **De façon plus générale, pour garantir une gestion correcte d'une section critique, il faut :**
  - Des primitives permettant de prendre un « verrou » exclusif (*lock*) en entrée de la section critique et de le relâcher en sortie (*unlock*)
  - Garantir l'indivisibilité de la séquence qui consulte et met à jour l'état d'un verrou
- **Comment faire ?**

Primitives de verrouillage

Cas d'un système monoprocesseur

- **Comment implémenter les primitives lock/unlock de façon indivisible ?**
- **Quelles sont les circonstances qui peuvent remettre en cause l'indivisibilité de l'opération ?**
  - Un appel synchrone à l'ordonnanceur ?
    - Non : pas de point de blocage entre la consultation et la modification de l'état du verrou
  - Un appel asynchrone à l'ordonnanceur ?
    - Oui : par défaut, un interruption matérielle peut survenir n'importe quand
- **On peut implémenter les primitives lock/unlock via des appels systèmes**
  - Idée : le noyau peut masquer les interruptions avant d'exécuter l'opération puis les démasquer ensuite

Primitives de verrouillage
Cas d'un système multiprocesseurs

- **La solution précédente est-elle valable dans ce contexte ?**
  - **Non** :
    - Les interruptions ne sont pas la seule cause d'entrelacements problématiques entre plusieurs flots d'exécutions
    - Plusieurs processeurs => parallélisme matériel : plusieurs instructions exécutées en parallèle

- **Comment assurer l'indivisibilité de la consultation et de la mise à jour d'une variable « verrou » malgré tout ?**
  - Impossible à garantir de façon générale/efficace en utilisant uniquement des techniques logicielles
  - Nécessité d'un support spécifique au niveau matériel :
    - Instruction « Test & Set » ou « Swap »
  - Technique également valable sur machine monoprocesseur

Instruction matérielle « Test & Set »

Définition

```
boolean
TestAndSet(boolean*target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

Utilisation pour réaliser l'exclusion mutuelle

```
do {

  while (TestAndSet(&lock))
    ; // do nothing

  // critical section
  ...

  lock = FALSE;

  // remainder section
  ...

} while (TRUE);
```

Instruction matérielle « Swap »

Définition

```
void
Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Utilisation pour réaliser l'exclusion mutuelle

```
do {
  key = TRUE;
  while (key == TRUE)
    Swap(&lock, &key);

  // critical section
  ...

  lock = FALSE;

  // remainder section
  ...

} while (TRUE);
```

Exclusion mutuelle basée sur « Test & Set »

Version qui satisfait le critère d'attente bornée

```
// data structures
// initially set to FALSE

boolean waiting[n];
boolean lock;
```

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // critical section
    ...

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;

    // remainder section
    ...

} while (TRUE);
```

Attente active et attente passive (1/2)

- **Attente active**
  - Principe des solutions étudiées jusqu'à présent :
    - Une demande de verrou est non bloquante
    - En cas d'échec, un processus réitère sa demande jusqu'à obtenir le verrou
  - Approche très inefficace sur un système monoprocesseur : à éviter systématiquement
  - Sur un système multiprocesseurs : approche potentiellement efficace dans certaines situations
    - Section critique très brève
    - Processeur mobilisé par l'attente active non réutilisable pour une autre activité

Attente active et attente passive (2/2)

- **Attente passive**
  - Une demande de verrou est bloquante :
    - Si le verrou est disponible, la demande est immédiatement satisfaite
    - Sinon :
      - Le processus demandeur est bloqué (mis en attente)
      - Et sera débloqué lorsque le verrou est disponible (ou, selon les cas, lorsque le verrou a une chance d'être disponible)
  - Solution plus efficace que l'attente active dans la plupart des cas

Mutex

- **Verrou simple à attente passive**
- **Primitives (pseudo-code)**
  - `void mutex_init(mutex_t *m)`
  - `void mutex_lock(mutex_t *m)`
    - Demande bloquante
  - `int mutex_trylock(mutex_t *m)`
    - Demande non bloquante
  - `void mutex_unlock(mutex_t *m)`
- **Variantes**
  - Vérification des erreurs (tentative de libérer un verrou déjà libre ou non détenu par le même processus ...)
  - Détection d'interblocage
  - Gestion récursive d'un verrou (ré-acquisition du même verrou au sein d'une section critique)

Cas d'étude :
gestion de parking N places (1/4)

```
int N = ... // nombre de places max

void entrer_parking() {
    while (N==0); // attendre
    N--;
}

void sortir_parking() {
    N++;
}
```

Code non synchronisé et donc FAUX

Cas d'étude : gestion de parking N places (2/4)

Synchronisation avec mutex

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

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

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

Incorrect : interblocage (monopolisation du verrou)

Cas d'étude : gestion de parking N places (3/4)

Synchronisation avec mutex

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

void entrer_parking() {
    mutex_lock(&m);
    while (N==0) {
        mutex_unlock(&m);
        mutex_lock(&m);
    }
    N--;
    mutex_unlock(&m);
}
```

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

Correct
(mais fastidieux et potentiellement inefficace)

Cas d'étude : gestion de parking N places (4/4)

- **Conclusion**
  - Les primitives de verrouillage simples ne sont pas adaptées pour la programmation aisée et efficace de tous les schémas de synchronisation
  - En particulier, on voit qu'il existe des besoins distincts :
    - Assurer que les données sont manipulées en exclusion mutuelle pour préserver leur cohérence
    - Bloquer/débloquer des processus en fonction de certaines conditions applicatives

Sémaphores (1/2)

- **Un autre mécanisme de synchronisation**
- **Un sémaphore correspond à un compteur (entier) synchronisé et une file d'attente**
- **Interface**
  - Initialisation du compteur **`count`** avec la valeur souhaitée
  - Méthode **`P`**
    - Autres noms : **`wait`**, **`sem_wait`**, **`down`** ...
    - Décrémente **`count`** et bloque l'appelant si **`count`** négatif
  - Méthode **`V`**
    - Autres noms : **`signal`**, **`sem_signal`**, **`up`** ...
    - Incrémente **`count`** et, si **`count`** négatif ou nul, réveille l'un des processus bloqués
    - Jamais bloquante
  - **Important** : Ces méthodes sont exécutées en exclusion mutuelle

Sémaphores (2/2)

```
typedef struct {
    int count;
    struct process *list; // list of blocked processes
} semaphore;
```

```
void sem_init(semaphore *s, int val) {
    s->count = val;
    s->list = NULL;
}
```

```
void P(semaphore *s) {
    s->count--;
    if (s->count < 0) {
        put(myself(), s->list);
        suspend();
    }
}
```

```
void V(semaphore *s) {
    s->count++;
    if (s->count <= 0) {
        struct process *p;
        p = get(s->list);
        wakeup(p);
    }
}
```

- **Rappel : méthodes exécutées en exclusion mutuelle**
- **Analogie avec un portillon à jetons**
  - On peut créditer des jetons à l'avance

Cas d'étude : gestion de parking N places avec sémaphore

```
int N = ... // nombre de places max
semaphore s;
...
sem_init(&s, N);
...

void entrer_parking() {
    P(&s);
}

void sortir_parking() {
    V(&s);
}
```

Cas d'étude : tampon producteur consommateur avec sémaphores (1/4)

- **On reprend l'exemple vu en introduction**
- **Tampon borné avec `BUFFER_SIZE` cases**

- **Processus producteur**
  - Peut déposer une donnée dans le tampon si le tampon n'est pas plein
- **Processus consommateur**
  - Peut récupérer une donnée dans le tampon si le tampon n'est pas vide
- **Tampon**
  - Tampon plein : 0 cases libres
  - Tapon vide : **`BUFFER_SIZE`** cases libres

Cas d'étude : tampon producteur consommateur avec sémaphores (2/4)

```
semaphore full; // supervision des cases pleines
semaphore empty; // supervision des cases vides
semaphore mutex; // exclusion mutuelle


// initialisations
sem_init(&full, 0);
sem_init(&empty, BUFFER_SIZE);
sem_init(&mutex,  1);
```

Cas d'étude : tampon producteur consommateur avec sémaphores (3/4)

Producteur

```
while (true) {
  // produce an item in
  // nextProduced

  P(&mutex);
  P(&empty);

  buffer[in] = nextProduced;
  in = (in + 1)%BUFFER_SIZE;
  count++;

  V(&mutex);
  V(&full);
}
```

Consommateur

```
while (true) {
  P(&full);
  P(&mutex);

  nextConsumed = buffer[out];
  out = (out + 1)%BUFFER_SIZE;
  count--;

  V(&mutex);
  V(&empty);

  // consume the item in
  // nextConsumed
}
```

Code INCORRECT !

Exercice : Trouver une séquence qui illustre le problème ...

Cas d'étude : tampon producteur consommateur avec sémaphores (4/4)

Producteur

```
while (true) {
  // produce an item in
  // nextProduced

  P(&empty);
  P(&mutex);

  buffer[in] = nextProduced;
  in = (in + 1)%BUFFER_SIZE;
  count++;

  V(&mutex);
  V(&full);
}
```

Consommateur

```
while (true) {
  P(&full);
  P(&mutex);

  nextConsumed = buffer[out];
  out = (out + 1)%BUFFER_SIZE;
  count--;

  V(&mutex);
  V(&empty);

  // consume the item in
  // nextConsumed
}
```

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)

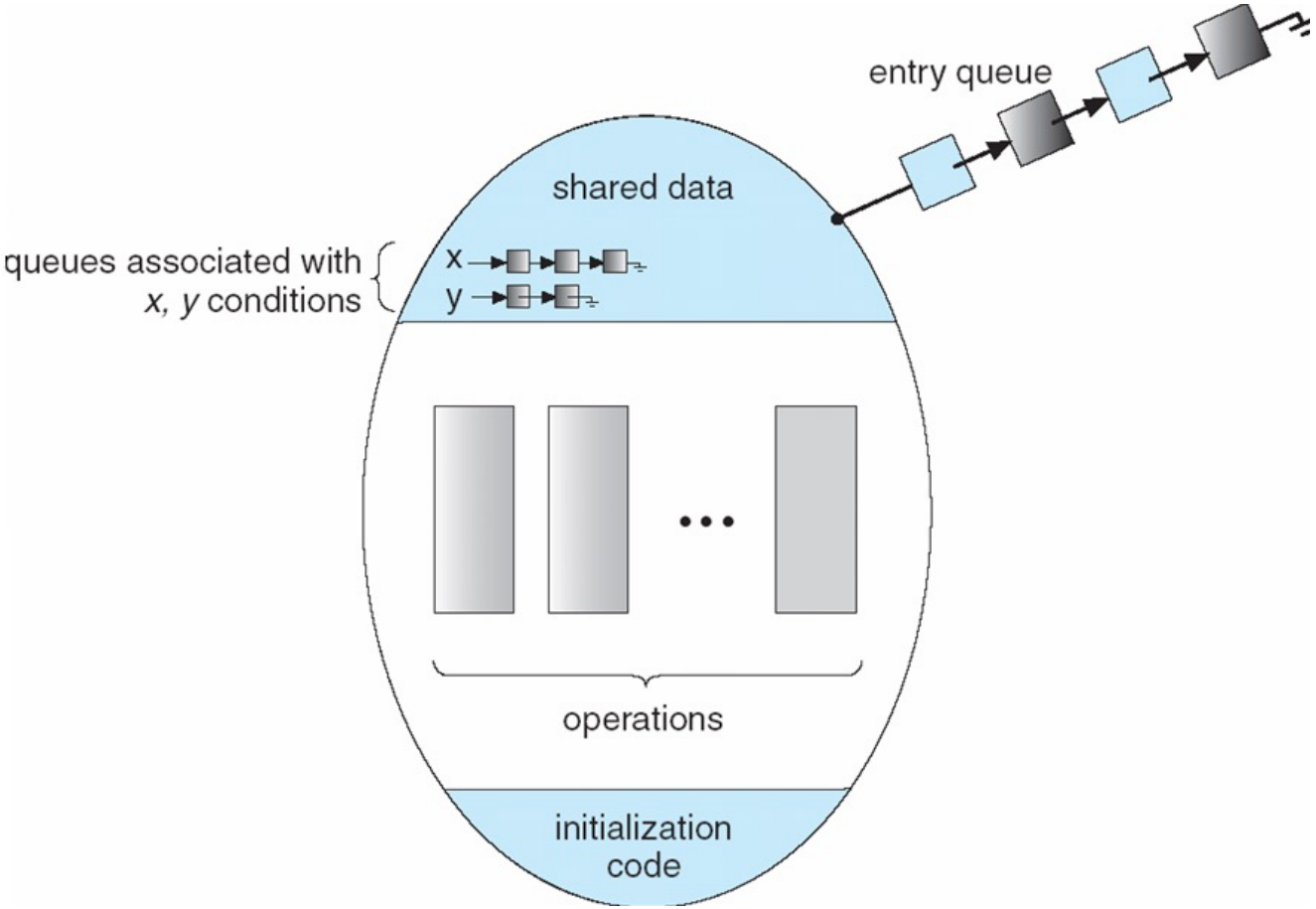
- **Variables de Conditions (suite)**
  - Une variable de condition fournit les opérations suivantes :
    - **`wait`** : bloque le processus appelant sur la variable de condition correspondante et libère le moniteur
    - **`signal`** (ou **`notify`**) : réveille l'un des processus bloqués sur la variable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

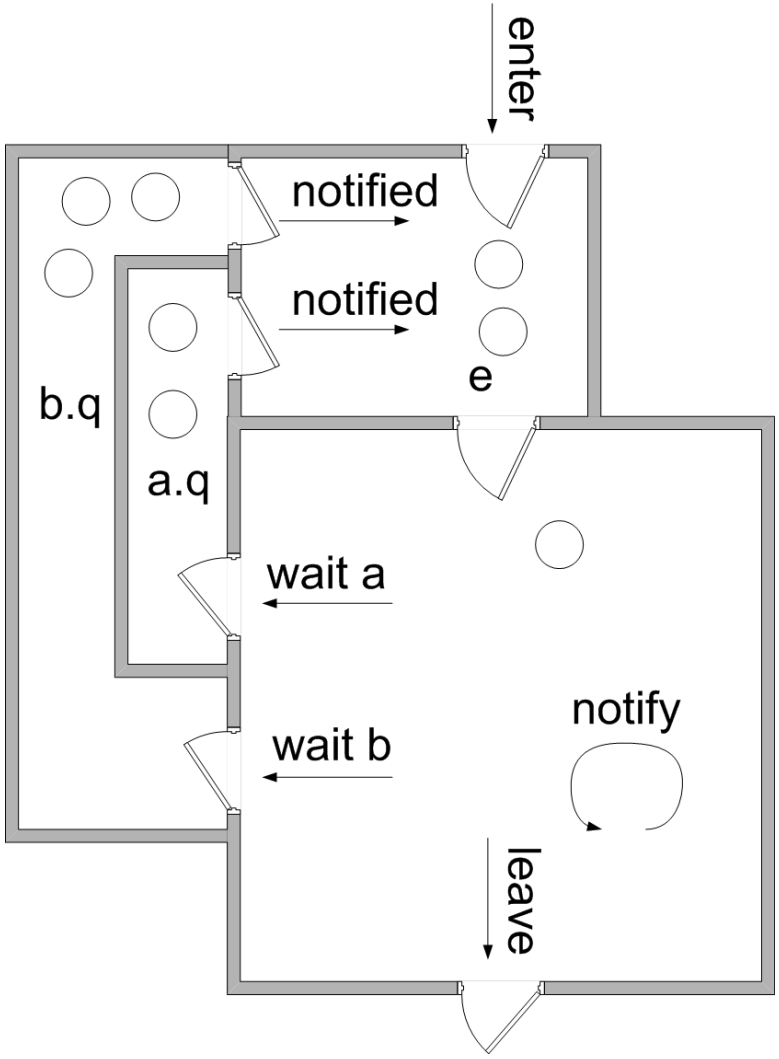
Moniteurs (7/7) - Synthèse des principales règles

- **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

Synchronisation : Pistes d'approfondissement

- Autres mécanismes de base
- Problèmes liés à la synchronisation
- Mémoires transactionnelles
- Retour sur les critères de progrès

Autres mécanismes élémentaires de synchronisation

- **Verrous lecteurs-rédacteurs (reader-writer locks)**
  - Observation : des accès concurrents à une donnée en lecture seule n'introduisent aucun conflit (= aucun risque)
  - Idée : différencier les demandes d'accès
    - Accès en lecture : possible malgré les accès concurrents (éventuels) d'autres lecteurs
    - Accès en écriture : un rédacteur doit manipuler les données en exclusion mutuelle (autre lecteurs ou rédacteurs interdits)
- **Barrières de synchronisation**
  - Scénario : N flots (threads/processus) concurrents (par exemple pour diviser une phase de calcul en N tâches.
  - Lorsqu'un flot a terminé sa tâche, il appelle la fonction *barrière*. Les (N-1) premiers flots sont bloqués lors de l'appel, le dernier flot débloque les autres et permet ainsi de passer à la phase suivante.
- **Ces mécanismes seront étudiés en TD/TP**

Problèmes liés à la synchronisation (1)
Inversion de priorité

- **Un processus P1 de haute priorité peut se retrouver bloqué en attente d'une ressource détenue (exclusivement) par un processus P2 de plus basse priorité**
- **Pire : P2 peut être préempté par un processus P3 de priorité intermédiaire (et ainsi de suite) => la durée du blocage de P1 est augmentée (voir infinie)**
- **Conséquences possibles**
  - Mauvaises performances, mauvaise réactivité du système
  - Dysfonctionnement (problèmes en cascades) – Exemple : robot Mars Pathfinder
  - Problème important dans les systèmes temps-réel
- **Solutions possibles**
  - N'utiliser que 2 niveaux de priorités
  - Elévation (temporaire) de la priorité d'un processus (ici P2)
    - Par héritage (en fonction des processus en attente)
    - Par tirage aléatoire (parmi le processus qui détiennent des verrous)

Problèmes liés à la synchronisation (2) Interblocages (deadlocks)

- **Définition du problème déjà vu dans le cours de bases de données**
- **Deux types d'approches pour gérer le problème**
  - Prévention
  - Détection et réaction
- **Prévention**
  - Technique 1 : Réservation globale des ressources à l'avance
  - Technique 2 : Respecter un ordre global sur les demandes de ressources
  - Technique 3 : Suivre les dépendances et empêcher la formation de cycles (refuser la demande fatale)
- **Détection**
  - Laisser l'interblocage survenir, le détecter et tuer un processus impliqué
  - Attention : contrairement aux SGBD, les systèmes d'exploitation généralistes n'ont pas de support intégré pour le « rollback » d'une tâche
- **Problème : ces solutions ne sont pas toujours applicables dans un système généraliste avec des applications arbitraires**
- **Variante du problème : livelocks**

Mémoires transactionnelles

- **Idée inspirée de la notion de *transaction* dans le domaine des bases de données**
  - Le programmeur doit simplement indiquer les frontières (début/fin) des sections critiques (= transactions)
  - Le programmeur n'a pas à déclarer les mécanismes utilisés (verrous …)
  - Le « système » sous-jacent se charge de gérer les problèmes de concurrence (généralement de manière optimiste)
- **Différentes niveaux d'implémentation possibles**
  - Logiciel, Matériel, Hybride
  - Support encore limité en production
- **Bénéfices**
  - Simplicité, composition aisée et correcte de code
- **Limitations**
  - Performances (pour certaines configurations)
  - Gestion des blocs de code ayant des effets de bord non annulables

Compléments sur la notion de *progrès*

- **Lors de la définition de la notion de section critique, nous avons vu défini la notion de *progrès*.**
- **En fait, selon les garanties fournies par un algorithme de synchronisation donné, on peut distinguer différents critères de progrès.**

- **Algorithmes bloquants (*blocking*):**
  - (à ne pas confondre avec les notions d'attente passive/active)
  - Un problème/retard/délai d'un thread peut éventuellement empêcher les autres threads de progresser
  - Les primitives de base que base que nous avons étudiées rentrent dans cette catégorie

- **Algorithmes non bloquants (*non-blocking*)**
  - Il existe différentes sous-catégories :
    - *Obstruction-free*, *Lock-free*, *Wait-free*

Compléments sur la notion de *progrès (suite)*

- ***Obstruction freedom*:** l'algorithme permet à un thread exécuté en isolation (c'est-à-dire pendant que tous les autres sont suspendus) pendant un nombre fini d'étapes de mener à bien son opération.

- ***Lock freedom*:** l'algorithme permet à au moins un thread de mener à bien son opération en un nombre fini d'étapes. Il est possible que certains threads doivent abandonner/réessayer leur opération.

- ***Wait freedom*:** l'algorithme garantit que chaque opération effectuée par chaque thread sera menée à bien en un nombre fini d'étapes.

Compléments sur la notion de *progrès (suite)*

- **Les algorithmes non bloquants ne sont pas sujet à des risques d'interblocages (deadlocks/livelocks) ou d'inversion de priorité.**

- **En matière de garanties de progrès, on a la hiérarchie suivante : blocking < obstruction free < lock free < wait free**
  - Cependant, en pratique, cette hiérarchie ne se transpose pas forcément au niveau des performances observées. Certains algorithmes non bloquants (en particulier *wait free*) peuvent être inefficaces.