Synchronisation

Menu

- Mise en contexte
- La problème section critique
- Synchronisation a niveau matériel
- Mutex et sémaphores
- Quelques problèmes classiques
- Moniteurs
- Exemples

Menu

- **Mise en contexte**
- La problème section critique
- Synchronisation a niveau matériel
- Mutex et sémaphores
- Quelques problèmes classiques
- Moniteurs
- Exemples

Mise en contexte

- Les processus et threads peuvent exécuter concurremment
  - Peut être interrompu à tout moment, en complétant partiellement l'exécution

- Les accès concurrents aux données partagées peuvent être incohérents

- Pour garantir la cohérence, il faut des mécanismes de synchronisation

Producteur-Consommateur (le retour)

```
while (true) {
        while (counter == BUFFER_SIZE) ;
        /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
     counter--;
}
```

Condition de course (Race Condition)

- `counter++`

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--`

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- **Résultat dépend de l'ordre d'exécution: condition de course**

Menu

- Mise en contexte
- **La problème section critique**
- Synchronisation a niveau matériel
- Mutex et sémaphores
- Quelques problèmes classiques
- Moniteurs
- Exemples

Section Critique

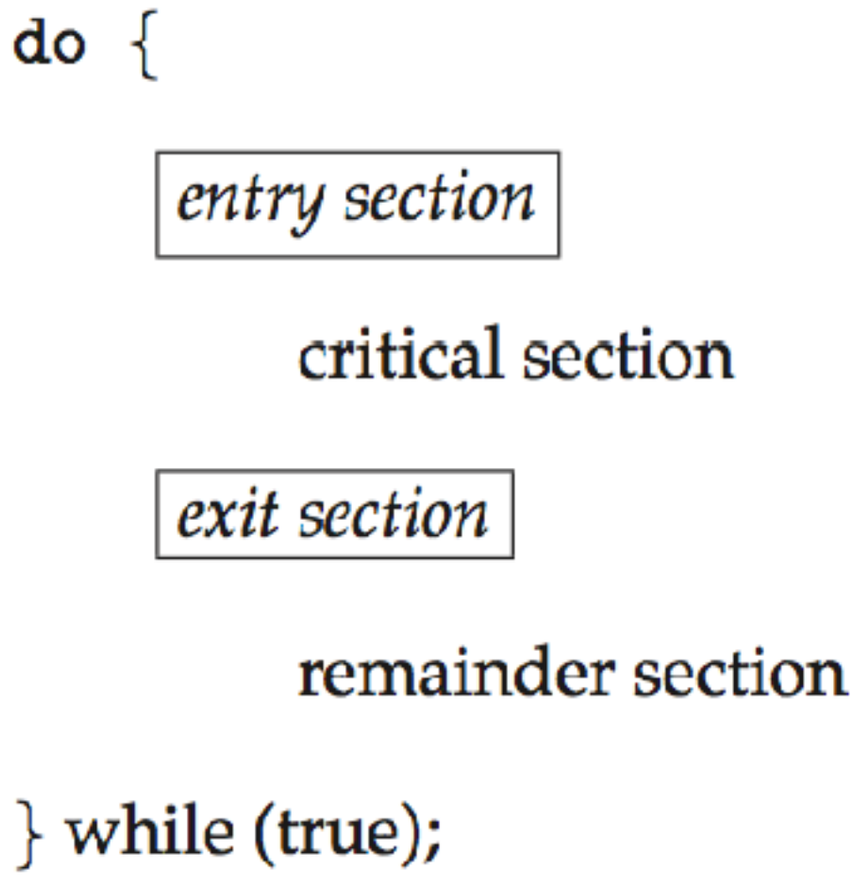
- Système avec ***n*** processus { $P_0, P_1, \ldots, P_{n-1}$ }

- Chaque processus a un segment de **section critique** de code
  - Le processus peut être en train de changer des variables communes, mettre à jour une table, écrire un fichier, etc.
  - Lorsqu'un processus est dans sa section critique, aucun autre ne peut être exécuté dans sa section critique

- Le **problème de la section critique** est de concevoir un protocole pour résoudre ce problème

Structure d'une Section Critique

- Chaque processus
  - doit demander la permission avant commencer la section critique dans la "**entry section**",
  - peut terminer la section critique avec une "**exit section**",
  - execute la reste du code (qui modifie les données locaux dans la "**remainder section**"



Solution au problème de la section critique

Une section critique devrait avoir les propriétés suivantes:

1. **Exclusion mutuelle** - Si un processus (thread) P_i est dans une section critique, aucun autre processus (thread) ne peut interférer

2. **Progrès** - Si aucun processus ne s'exécute dans sa section critique et qu'il existe des processus qui souhaitent entrer dans leur section critique, seuls les processus qui ne sont pas dans leur section restante (remainder section) peuvent participer à la sélection du processus qui entrera dans sa section critique suivante; la sélection ne peut pas être reportée indéfiniment

3. **Attente limitée** - Une limite doit exister sur le nombre de fois que les autres processus sont autorisés à entrer dans leurs sections critiques après qu'un processus a fait une demande pour entrer dans sa section critique et avant que cette demande soit accordée

    - Supposons que chaque processus s'exécute à une vitesse non nulle

    - Aucune hypothèse concernant la vitesse relative des n processus

Section critique dans le Noyau

- Deux approches pour gérer les sections critiques dans les systèmes d'exploitation, selon que le noyau est préemptif ou non préemptif
  - **Préemptif** - permet la préemption du processus lors de l'exécution en mode noyau
    - Particulièrement difficile dans les architectures multiprocesseurs, mais cela rend le système plus réactif
  - **Non-préemptif** - s'exécute jusqu'à ce qu'il quitte le mode noyau, bloque ou donne volontairement du CPU
    - Essentiellement dépourvu de conditions de concurrence sur les structures de données du noyau en mode noyau, car un seul processus actif dans le noyau à la fois

Solution de Peterson

- Solution restreinte à deux processus en exécution alternative (critical section et remainder section)
- Assumer que le **`load`** et le **`store`** sont des opérations **atomique**
- Les deux processus partagent deux variables:
  - **`int turn`**
  - **`boolean flag[2]`**
- Le variable **`turn`** indique à quel tour il faut entrer dans sa section critique
- The **`flag`** array est utiliser pour indiquer si un processus est prêt à entrer dans la section critique

  `flag[i] == true` implique que P_i est prêt

Algorithm for Process P_i

i = moi, j = autre

si `j` est prêt, il devrait entrer dans sa section critique

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j); // wait
                // critical section
        flag[i] = false;
                // remainder section
 } while (true);
```

- Peut prouver que:
  1. Exclusion mutuelle est conservé
  2. Progrès est satisfait
  3. Attente limite est satisfait

Support Matériel

- De nombreux systèmes fournissent un support matériel pour le code de section critique
- Toutes les solutions ci-dessous basées sur l'idée de **verrouillage (locking)**
  - Protéger les régions critiques via des verrous
- Uniprocessors - pourrait désactiver les interruptions
  - Le code courant s'exécuterait sans préemption
  - Généralement trop inefficace sur les systèmes multiprocesseurs
    - Les systèmes d'exploitation qui l'utilisent ne sont pas extensibles
- Les machines modernes fournissent des instructions spéciales sur le matériel atomique
  - **Atomique = non interruptible**
    - `test_and_set()`
    - `compare_and_swap()`

Solution à l'aide de verrous

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```

Solution avec test_and_set()

- Deux `test_and_set()` ne peut pas être exécuté simultanément (ils sont exécutés atomiquement)
  - Même sur les multiprocesseurs (ils seront commandés pour éviter l'exécution simultanée)
- Variable partager **`lock`**, initialisé à FALSE

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

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */
   /* critical section */
   lock = false;
   /* remainder section */
} while (true);
```

compare_and_swap()

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;
   if (*value == expected)
      *value = new_value;
   return temp;
}
```

Solution avec compare_and_swap()

- Variable partager **`lock`** initialisé à FALSE

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)   //value,expected,new
       ; /* do nothing */
       /* critical section */
    lock = 0;
       /* remainder section */
} while (true);
```

- Exclusion mutuelle - oui
- Progrès - oui
- Attente limité - non

Attente limitée avec test_and_set()

Partager

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);  //only first lock==false will set key=false
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;    //look for the next P[j] waiting: bound-waiting req.
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false; // nobody waiting
   else
      waiting[j] = false; //wakeup only one process P[j] without releasing
lock
   /* remainder section */
} while (true);
```

Menu

- Mise en contexte
- La problème section critique
- Synchronisation a niveau matériel
- **Mutex et sémaphores**
- Quelques problèmes classiques
- Moniteurs
- Exemples

Mutex Locks

- Les solutions précédentes sont compliquées et généralement inaccessibles aux programmeurs d'application
- Les concepteurs du SE construisent des outils logiciels pour résoudre un problème de section critique
- La synchronisation la plus simple est faite avec les **mutex**
- Protégez les régions critiques par `acquire()` et `release()`
  - Variable booléenne indiquant si le verrou est disponible ou non

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;
}
release() {
   available = true;
}
```

```
do {
   acquire()
      critical section
   release()
      remainder section
} while (true);
```

Mutex Locks

- Appels a **`acquire()`** and **`release()`** doivent être atomique

- Désavantages - “**busy waiting**”
  - Tous les autres processus essayant d'obtenir le verrou doivent continuellement boucler
  - Ce verrou est donc appelé un **spinlock**
  - Très gaspillage de cycles CPU
  - Pourrait être encore plus efficace que les changements de contexte (coûteux) pour des temps d'attente plus courts

Sémaphore

- Sémaphore ***S*** – Valeur entière
- Deux opérations standard sur ***S*** : `wait()` et `signal()`

```
wait (S) {
    while (S <= 0); // busy wait
    S--;
}
```

```
signal (S) {
    S++;
}
```

Usage de sémaphores

- **Counting semaphore** – valeur entière interprétable comme nombre de ressources disponibles
- **Binary semaphore** – 0 ou 1
  - Équivalent à un **mutex lock**
- Peut résoudre divers problèmes de synchronisation
- E.g. garantir que processus $\mathbf{P_1}$ exécute $\boldsymbol{S_1}$ avant que $\mathbf{P_2}$ exécute $\boldsymbol{S_2}$

```
synch = 0
P1:
   S1;
   signal(synch);    //sync++ has added one resource
P2:
   wait(synch);      //executed only when synch is > 0
   S2;
```

Implémentation de sémaphores

- Doit garantir que deux processus ne peut pas exécuter **`wait()`** et **`signal()`** sur le même sémaphore en même temps
- Ainsi, la *implémentation devient le problème de section critique*, où les codes d'attente et de signal sont placés dans la section critique
  - Pourrait maintenant avoir **busy waiting** dans la implémentations de la section critique
    - Mais le code d'implémentation est court
    - Peu de **busy waiting** si la section critique est rarement occupée
- Notez que les applications peuvent passer beaucoup de temps dans des sections critiques et que ce n'est pas une bonne solution
- Une solution est ce que le processus se bloque lui-même au lieu d'entrer en attente en attente

Pas de “Busy Waiting”

- Avec chaque sémaphore, il y a une file d'attente associée
- Chaque entrée dans une file d'attente a deux éléments de données:
  - Valeur (de type entier)
  - Pointeur vers l'enregistrement suivant dans la liste
- Deux opérations:
  - **`block()`** - place le processus appelant l'opération dans la file d'attente appropriée
  - **`wakeup()`** - supprime l'un des processus dans la file d'attente et le place dans la file d'attente prête

```
typedef struct {
   int value;
   struct process *list;
} semaphore;
```

```
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}
```

Interblocage (Deadlock)

- **Interblocage** – deux ou plusieurs processus attendent indéfiniment un événement qui peut être causé par un seul processus en attente
- Ex: *s* et *Q* sont deux *semaphores initialisé à 1*

P_0	P_1
wait(S); //exec 1st	**wait(Q);** // exec 2nd
wait(Q); //exec 3rd	**wait(S);** // exec 4th
...	...
signal(S);	**signal(Q);**
signal(Q);	**signal(S);**

- Plus de verrous -> plus haute risque d'interblocage

Famine (Starvation)

- **Famine (starvation)** – **blocage indéfini**
  - Se produit lorsqu'un thread n'a jamais l'opportunité de progresser
  - Peut être causé par un "interblocage" ou un "live deadlock", ou toute sorte d'autres circonstances indésirable

Menu

- Mise en contexte
- La problème section critique
- Synchronisation a niveau matériel
- Mutex et sémaphores
- **Quelques problèmes classiques**
- Moniteurs
- Exemples

Problèmes classiques de synchronisation

- Problèmes classiques utilisés pour tester les schémas de synchronisation nouvellement proposés
  - Problème des producteurs/consommateurs (Bounded-Buffer)
  - Problème des Lecteurs et Écrivains
  - Problème des Dining-Philosophers

Producteurs / Consommateurs

- Les données partagés:
  - ***buffer*** de grandeur ***n***
  - Semaphore **`mutex`** initialisé a 1 (pour assurer l'exclusion mutuelle dans buffer)
  - Semaphore **`full`** initialisé a 0 (pour indiquer le numéro de "choses" dans le buffer)
  - Semaphore **`empty`** initialized to the value *n* (pour indiquer le numéro d'espaces vides dans le buffer)

Producteurs / Consommateurs

- La structure du “Producteur”

```
do {
        ...
         /* produce an item in
     next_produced */
        ...
     wait(empty);
     wait(mutex);
        ...
         /* add next_produced to the
      buffer */
        ...
     signal(mutex);
     signal(full);
} while (true);
```

- La structure du “Consommateur”

```
do {
    wait(full);
    wait(mutex);
        ...
     /* remove an item from buffer to
     next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
     /* consume the item in next_consumed
     */
        ...
} while (true);
```

Lecteurs-Écrivains

- Une donnée est partagée entre plusieurs threads
  - Lecteurs – Certains threads n'y accèdent qu'en lecture
  - Écrivains – Certains threads y accèdent en lecture et écriture
- Problème:
  - Permettre plusieurs accès simultanées en lecture
  - Exclusion mutuelle en cas d'accès en écriture
- 2 variations:
  - *Première variante* - lecteur attend seulement quand l'écrivain a l'autorisation d'utiliser un objet partagé
  - *Deuxième variante* - quand l'écrivain est prêt, il effectue une écriture dès que possible
  - *Les deux peuvent causer la famine:*
    - *première:* les lecteurs continuent d'arriver alors que les écrivains ne sont jamais traités
    - *deuxième:* les écrivains continuent d'entrer tandis que les lecteurs ne sont jamais traités
  - Le problème est résolu sur certains systèmes par le noyau fournissant des **verrous de lecteur-graveur** (reader-writer locks)

Lecteurs-Écrivains

- Solution pour première variante (aucun lecteur attente)
- Données partagées
  - Semaphore **`rw_mutex`** initialisé à 1 (utiliser pour accès exclusive au données)
  - Semaphore **`mutex`** initialisé à 1 (utiliser pour accès exclusive au variable **`read_count`**)
  - Variable entier **`read_count`** initialisé à 0 (numéro de processus en train de faire du lecteur)

Lecteurs-Écrivains

- La structure de l'écrivain:

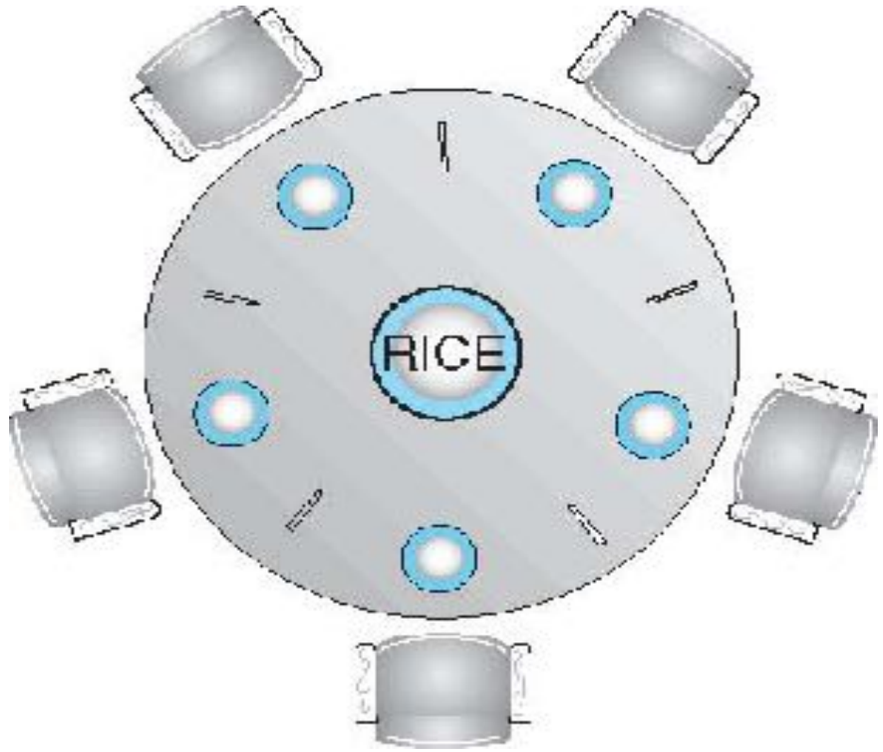
```
do {
        wait(rw_mutex);
        ...
         /* writing is performed */
        ...
        signal(rw_mutex);
} while (true);
```

- La structure du lecteur:

```
do {
     wait(mutex);
     read_count++;
     if (read_count == 1)
       wait(rw_mutex);
     signal(mutex);
       ...
     /* reading is performed */
       ...
     wait(mutex);
     read_count--;
     if (read_count == 0)
       signal(rw_mutex);
     signal(mutex);
} while (true);
```

Dining Philosophes

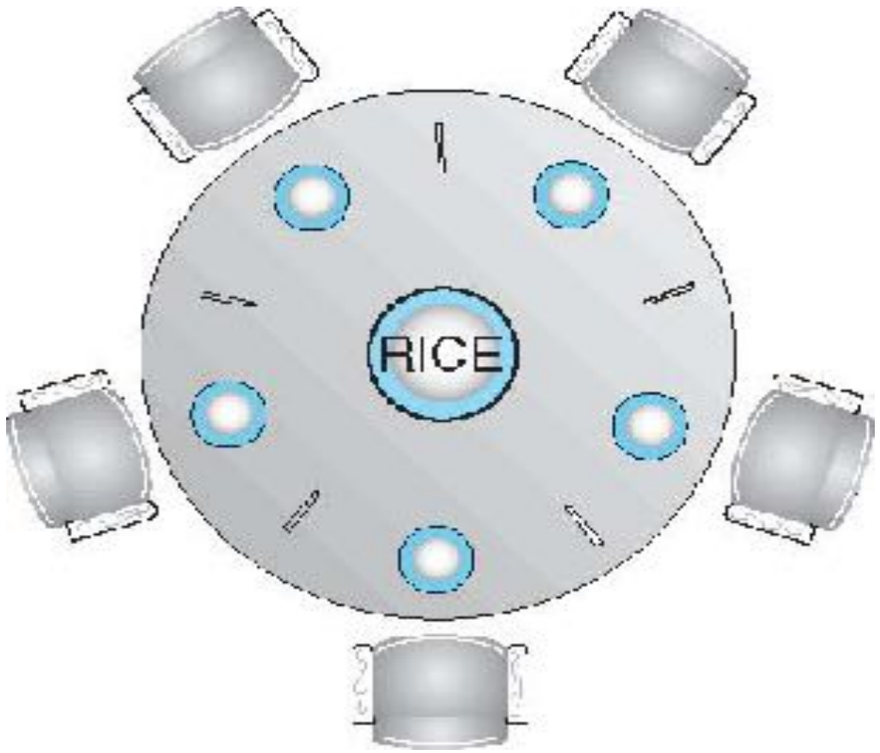
- Les philosophes passent leur vie à penser et à manger
- Ne peut pas interagir entre eux
  - Besoin de deux chopsticks (baguette) pour manger
  - Remplace les chopsticks après avoir fini
- Données partagées:
  - Bowl of rice (data set)
  - Sémaphore **`chopstick[5]`** initialisé à 1



Dining Philosophes

- La structure du philosophe ***i***

```
do   {
        wait(chopstick[i]);
        wait(chopstick[(i+1)%5]);
        //  eat
        signal(chopstick[i]);
        signal(chopstick[(i+1)%5]);
        //  think
} while (TRUE);
```



- Quel est la problème içi?

Dining Philosophes

- Solutions possibles:
  - Maximum de quatre philosophes à une table de cinq et plus
  - Ramasser des chopsticks seulement si les deux sont disponibles
  - les philosophes impaires choisissent la première gauche, les philosophes pairs choisissent le droite
- Une solution sans impasse ne suffit pas, la solution doit aussi assurer qu'aucun philosophe ne meurt de faim

Menu

- Mise en contexte
- La problème section critique
- Synchronisation a niveau matériel
- Mutex et sémaphores
- Quelques problèmes classiques
- **Moniteurs**
- Exemples

Difficultés liées aux mutex et sémaphores

- Très difficile a débogguer, e.g. :
  - Order inversé: **`signal(mutex)`** ... **`wait(mutex)`**
  - Appels répété: **`wait(mutex)`** ... **`wait(mutex)`**
  - Manque d'appel: **`wait(mutex)`** or **`signal(mutex)`** (or both)

- Peut très facilement résulté en interblocage ou famine

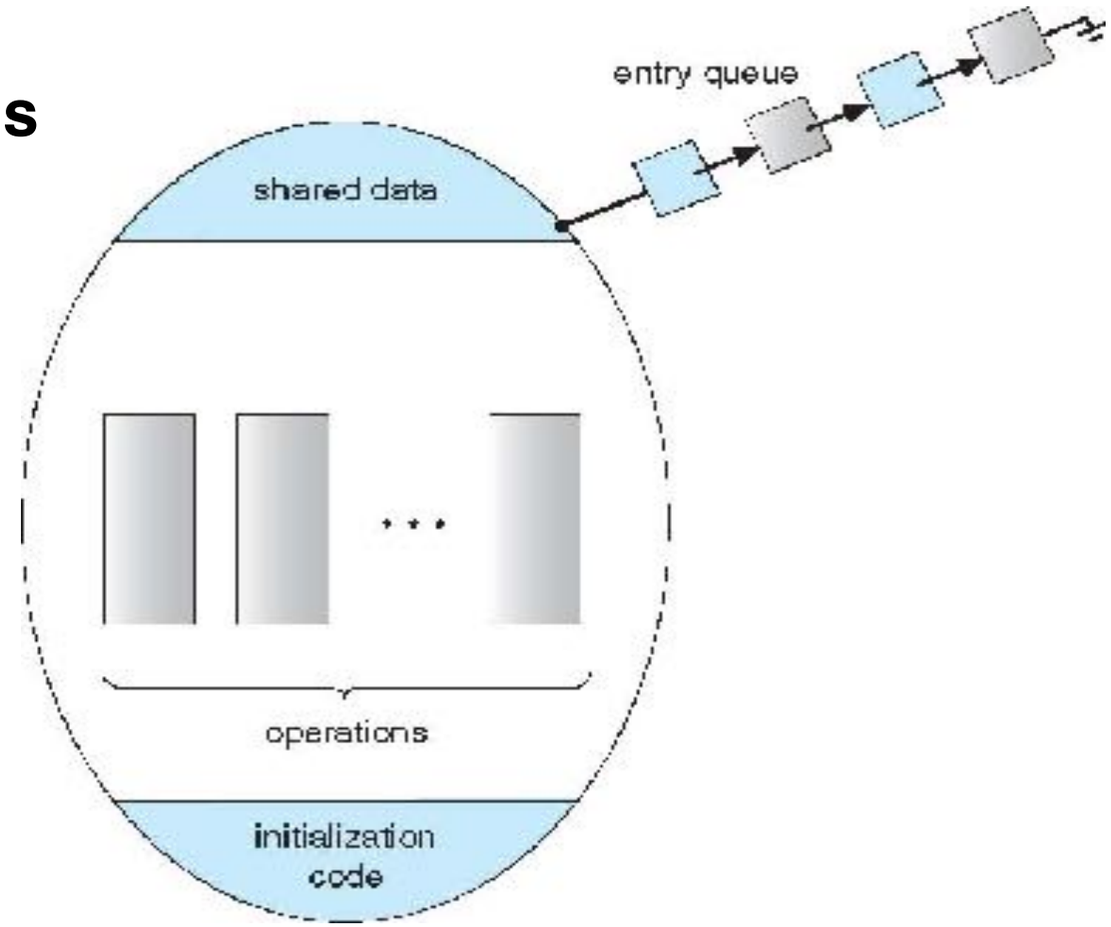
- Une solution proposée est à un niveau d'abstraction supérieure: un type moniteur

Moniteurs

- Une abstraction de plus haut niveau qui fournit un mécanisme pratique et efficace pour la synchronisation des processus
- **Un seul processus peut être actif dans le moniteur à la fois**

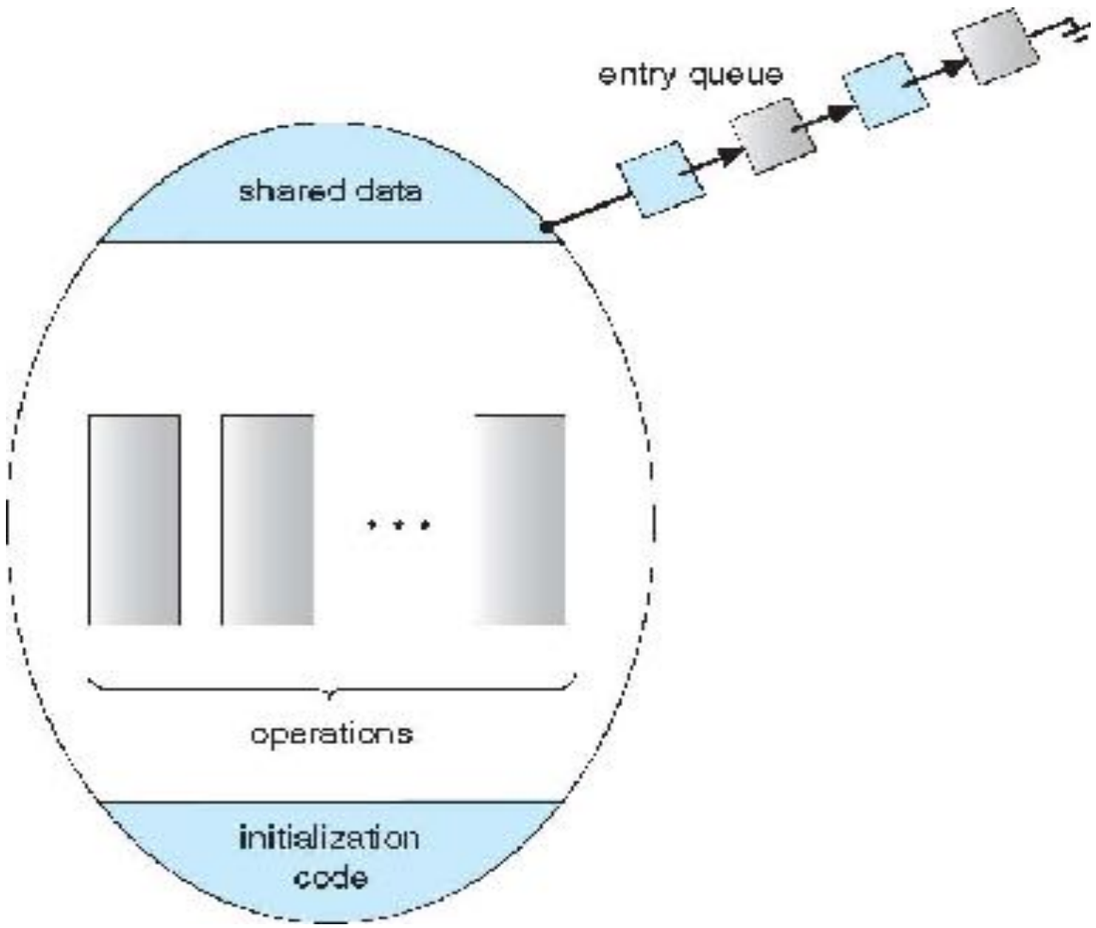
```
monitor monitor-name
{
   // shared variable
   declarations
   function P1(...) { ... }
   ...
   function Pn(...) { ... }

   initialization_code (...)
   { ...}
}
```



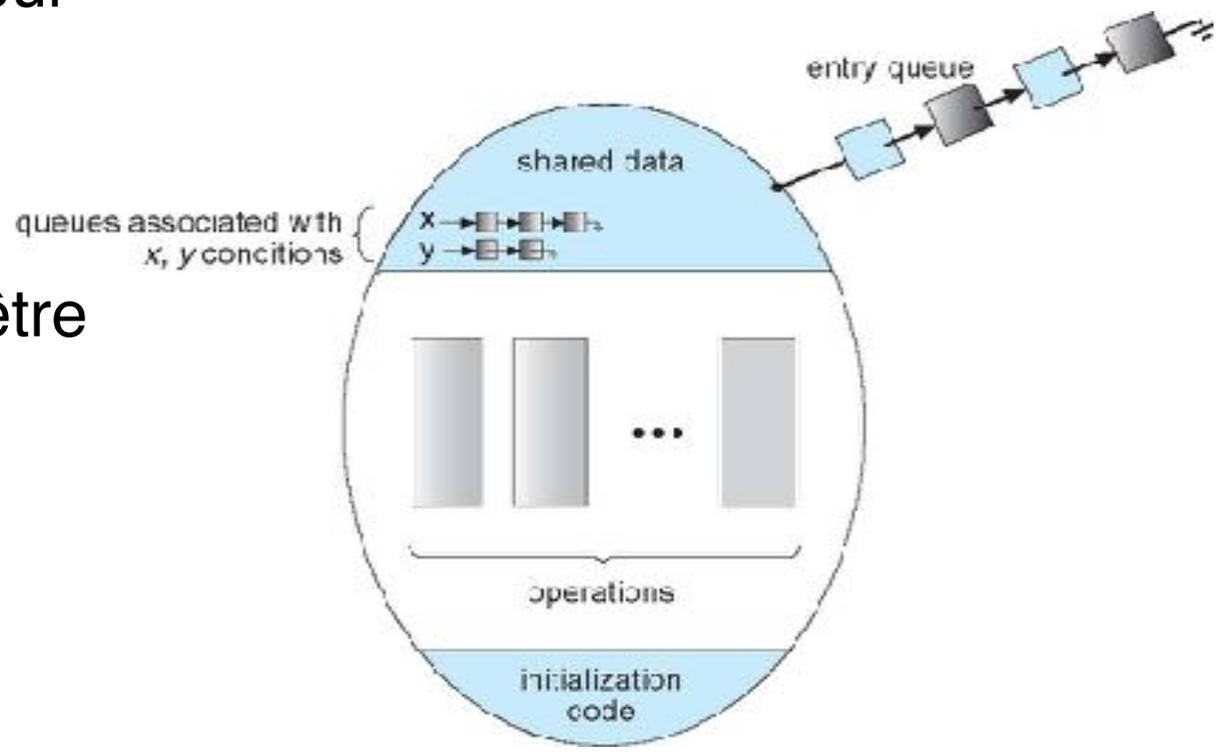
Moniteurs

- C’est facile mais qu’est qu’on fait si un processus veut attendre pour un resource **dans** un moniteur?
- Ex: Le consommateur attend pour quelque chose a consommer
- Solutions: Les conditions



Variables de condition

- Mécanisme de synchronisation pour les moniteurs déclarés avec les variables:
  - `condition x, y;`
- Seules deux opérations peuvent être appelées sur une variable de condition:
  - **`x.wait()`** – le processus est bloquer jusqu'à un **`x.signal()`**
  - **`x.signal()`** – recommence un processes qui attend avec un **`x.wait()`**
    - Si pas de **`x.wait()`**, fait rien (différent que le cas avec sémaphores)



Variables de condition

- Si processus **P** fait **`x.signal()`**, avec **Q** en état **`x.wait()`**, ce qui se passe ensuite?
  - Si Q est repris, alors P doit attendre, sinon deux processus seront actifs dans le moniteur

- Options:
  - **Signal and wait** – P attend jusqu'à ce que Q quitte le moniteur ou attende une autre condition
  - **Signal and continue** – Q attend jusqu'à ce que P quitte le moniteur ou attende une autre condition

Solution aux Dining Philosophers

```
monitor DiningPhilosophers
{
    enum { THINKING; HUNGRY, EATING)
    state[5];
    condition self[5];

    void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
           self[i].wait();
    }

    void putdown(int i) {
      state[i] = THINKING;
      // test left and right neighbors
      test((i+4)%5);
      test((i+1)%5);
    }
```

```
    void test(int i) {
      if ((state[(i+4)%5] != EATING) &&
           (state[i] == HUNGRY) &&
         (state[(i+1)%5] != EATING) ) {
           state[i] = EATING;
           self[i].signal();
      }
    }

    initialization_code() {
      for (int i = 0; i < 5; i++)
       state[i] = THINKING;
    }
} // end monitor
```

Solution aux Dining Philosophers

- Chaque philosophe *i* appelle les opérations pickup () et putdown () dans l'ordre suivant:

```
DiningPhilosophers.pickup(i);
...
EAT
...
DiningPhilosophers.putdown(i);
```

- Pas de interblocage, mais le famine est possible

Implémentation des Moniteurs

- Variables

```
semaphore mutex;  // (initialisé à 1 - contrôle l'access au moniteur)
semaphore next;   // (initialisé à 0 - utilisé pour redonner le contrôle a un processus qui a été bloquer après un x.signal)
int next_count = 0; // (nombre de processus suspendus par les conditions)
```

- Chaque fonction ***F*** vais être remplacer par:

```
wait(mutex);
...
body of F;
...
if (next_count > 0)
    signal(next);
else
    signal(mutex);
```

- L'exclusion mutuelle au sein d'un moniteur est assurée

Implémentation des Moniteurs

- Pour chaque variable condition ***x***:

```
semaphore x_sem;   // (initialisé à 0)
int x_count = 0;
```

- L'opération **x.wait()** :

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

- L'opération **x.signal()** :

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
// else fait rien
```

Reprise de processus dans un moniteur

- Si beaucoup de processus attend pour **`x`**, et **`x.signal()`** est exécuter, quel processus est repris?

- C'est encore un question d'ordonnancement

- **conditional-wait** est construit avec la forme **`x.wait(c)`**
  - **`c`** est le **numéro priorité**

Un moniteur pour allouer une seule ressource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time); // time: temps maximum qu'il gardera la ressource
        busy = TRUE;
    }

    void release() {
        busy = FALSE;
        x.signal();
    }

    initialization_code() {
        busy = FALSE;
    }
}
```

Inversion de Priorité

- Problème d'ordonnancement lorsque le processus de priorité inférieure contient un verrou requis par un processus de priorité supérieure

- Scenario :
  - Processes avec des priorités **L** < **M** < **H**
  - **L** est donné un resource *r*
  - **H** veut le resource et est prévu pour le prochain accès
  - Avant que L est terminé, **M** génère une interruption est prend *r*

- Pas un problème avec seulement deux niveaux de priorités, mais deux niveaux sont insuffisants pour la plupart des systèmes d'exploitation
- Solution: **priority-inheritance protocol**
  - Tous les processus de priorité inférieure avec une ressource demandée par un processus de priorité supérieure héritent du niveau de priorité supérieur jusqu'à ce qu'il libère la ressource

Menu

- Mise en contexte
- La problème section critique
- Synchronisation a niveau matériel
- Mutex et sémaphores
- Quelques problèmes classiques
- Moniteurs
- **Exemples**

Synchronization Examples

- Windows XP
- Solaris
- Linux
- Pthreads

Windows XP

- Utilise des masques d'interruption pour protéger l'accès aux ressources globales sur les systèmes monoprocesseur

- Utilise les **spinlocks** sur les systèmes multiprocesseurs
  - Spinlocking-thread ne sera jamais préempté

- Fournit également des **objets dispatcher** (en dehors du noyau) qui peuvent servir de verrous de mutex, de sémaphores, d'événements et de minuteurs
  - **Événement** s'agit comme une variable de condition (notifier les threads lorsque la condition se produit)
  - **Minuteur** notifie un ou plusieurs threads lorsque la durée spécifiée a expiré
  - Les objets Dispatcher ont un **état signalé** (objet disponible) ou un **état non signalé** (le thread bloquera)

Solaris

- Implémente une variété de verrous pour prendre en charge le multitâche, le multithreading (y compris les threads en temps réel) et le multitraitement
- Utilise des **verrous mutex adaptatifs** pour une meilleure efficacité lors de la protection des données à partir de segments de code court
  - Commence comme un verrou de sémaphore spin-lock
  - Si le verrouillage est maintenu, et par un thread s'exécutant sur un autre CPU, tourne
  - Si le verrouillage est maintenu par un thread non opérationnel, bloquez et dormez, attendez que le signal de verrouillage soit libéré
- Utilise les **variables de condition**
- Utilise des verrous de **readers-writers** lorsque des sections de code plus longues ont besoin d'accéder aux données
- Utilise des **turnstiles** pour ordonner la liste des threads en attente d'acquisition d'un mutex adaptatif ou d'un verrou reader-writer
- L'héritage de priorité par turnstile donne au thread en cours le plus haut des priorités des threads dans son turnstile

Linux

- Linux:
  - Avant la version 2.6 du noyau, désactive les interruptions pour implémenter des sections critiques courtes
  - Version 2.6 et plus, complètement préemptif

- Linux fournit:
  - sémaphores
  - spinlocks
  - versions readers-writers des deux

- Sur un système à processeur unique, les spinlocks sont remplacés en activant et en désactivant la préemption du noyau

Pthreads

- L'API Pthreads est indépendante du système d'exploitation
- Il offre:
  - mutex
  - variables de condition
- Les extensions non portables incluent:
  - verrous en lecture-écriture
  - spinlocks

Sommaire

- Quand beaucoup de processus accès les mêmes données on peut avoir des conditions de courses
- A niveau matériel, on a des opérations “atomique”
- Les mutex et sémaphores sont utiliser pour contrôler l’accès au données partagées
- Les moniteurs sont un types de données plus haute niveau pour partagé les données
- Les problèmes classique sont la producteur/consommateur (bounded buffer), lecteur-écrivain, et “dining philosophe”