Systèmes d'exploitations II

Présentation #7 : Les moniteurs et autres mécanismes de synchronisation

12/12/2021

Ahmed Benmoussa

Centre universitaire d'Aflou

Rappel: Les sémaphores

- Variable globale entière non négative.
- Manipulés par deux opérations: **P()** et **V()**.
- **Utilisation**: Exclusion mutuelle et ordonnancement des évènements.

Rappel: Les sémaphores

```
Semaphore fullSlots = 0;          // initialement, pas de boisson
Semaphore emptySlots = bufSize;
                                  // initialement, buffer vide
Semaphore mutex = 1;              // aucun n'utilise la machine

  Producer(item) {
      semaP(&emptySlots);         // attendre liberation espace
      semaP(&mutex);              // attendre liberation machine
      Enqueue(item);
      semaV(&mutex);
      semaV(&fullSlots);          // informer consommateur
                                  // l'existence de boisson
  }
  Consumer() {
      semaP(&fullSlots);          // verifier boissons
      semaP(&mutex);              // attendre liberation machine
      item = Dequeue();
      semaV(&mutex);
      semaV(&emptySlots);         // informer producteur besoin
      return item;
```

fullSlots signalle boissons

Proteger l'integrité de la section critique avec **mutex**

emptySlots signalle espace libre

Scénario

- Supposant les deux opérations down du producteur sont inversés.
- Et que le buffer est plein.

Problème de sémaphores

```
Semaphore fullSlots = 0;
Semaphore emptySlots = bufSize;

Semaphore mutex = 1;

  Producer(item) {
       semaP(&mutex);
       semaP(&emptySlots);
       enqueue(item);
       semaV(&mutex);
       semaV(&fullSlots);

  }
  Consumer() {
       semaP(&fullSlots);
       semaP(&mutex);
       item = Dequeue();
       semaV(&mutex);
       semaV(&emptySlots);
       return item;
  }
```

On doit être très prudent lors de l'utilistation des sémaphores!

Les moniteurs (*Monitors*)

- Brinch Hansen (1973) et Hoare (1974) ont proposé un méchanisme de synchronisation haut-niveau appelés: **Moniteur (*Monitor*).**
- Utilisés afin de faciliter l'écriture des programmes.
- Un Moniteur est un ensemble de procédures, variables, et structure de données.
- Les moniteurs ne sont pas implémentés en langage C.

Les moniteurs (*Monitors*)

- Un seul processus peut être actif a l'intérieur d'un moniteur.
- Peut être utilisé pour une section critique.
- Les appels aux procédures d'un moniteur diffèrent des autres procédures.
- Quand un processus appelle une procédure moniteur, les premières instructions vérifie si un autres processus exécute cette procédure.

Les moniteurs (*Monitors*)

- Utilise les **variables de condition** avec deux opération: ***Wait*** et ***Signal*** pour bloquer les processus qui ne peuvent pas continuer l'exécution.
- Les variables de condition ne sont pas des compteurs.

Les moniteurs: Pseudo-code

```
monitor ProducerConsumer
 condition full, empty;
 integer count;
  procedure insert(item: integer);
  begin
   if count = N then wait(full);
      insert item(item);
      count := count + 1;
      if count = 1 then signal(empty)
   end;
  function remove: integer;
  begin
   if count = 0 then wait(empty);
      remove = remove item;
      count := count - 1;
      if count = N - 1 then signal(full)
   end;
 count := 0;
end monitor;
```

```
procedure producer;
begin
 while true do
    item = produce item;
    ProducerConsumer.insert(item)
 end
end;

procedure consumer;
begin
 while true do
     item = ProducerConsumer.remove;
     consume item(item)
 end
end;
```

Les moniteurs

- Quel est la différence entre (**`Wait, Signal`**) et (**`Sleep, Wakeup`**)?

- Les procédures a l'intérieur des moniteurs finissent l'exécution de **`Wait`** avant que l'ordonnanceur switche vers un autre processus.

Les moniteurs sous Java

- Le language Java utilise **`synchronized`** pour déclarer les méthodes concurrentes.
- Les méthode déclarés avec `synchronized` sont exécutés par un et un seul thread a la fois.

Les moniteurs sous Java

```
static class producer extends Thread {
  public void run( ) {
   int item;
   while (true) {item = produce item( );
      mon.inser t(item);
   }
  }
 private int produce item( ) { ... } // produire
}

static class consumer extends Thread {
  public void run( ) {
   while (true) {
      item = mon.remove( );
      consume item (item);
   }
  }
 private void consume item(int item) { ... }// consommer
}
```

```
static class our_monitor {
  private int buffer[ ] = new int[N];
  private int count = 0, lo = 0, hi = 0;

  public synchronized void insert(int val) {
   if (count == N) go to sleep( );
   buffer [hi] = val;
   hi = (hi + 1) % N;
   count = count + 1;
   if (count == 1) notify( );
  }
  public synchronized int remove( ) {
   int val;
   if (count == 0) go to sleep( );
   val = buffer [lo];
   lo = (lo + 1) % N;
   count = count – 1;
   if (count == N – 1) notify( );
  return val;
  }
private void go_to_sleep() { try{wait();}
catch(InterruptedException exc) {};}
}
```

Les moniteurs

- Les moniteur sont un concept de programmation.
- Les moniteurs sont utilisable pour un nombre réduit de langage de programmation.
- Ces mécanisme de synchronisation sont inutilisable sur des systèmes distribués avec multitude de CPU ou chacun a sa propre mémoire.
- Les sémaphores et les moniteurs ne permettent pas l'échange de messages.

- **On a besoin d'autres mécanismes**.

Echange de messages (Message Parsing)

- Cette méthode de communication interprocessus utilise deux primitives: **envoyer (*send*)** et **recevoir (*receive*)**.
- Ces primitives sont des appels système (comme les sémaphores).

```
send(destination, &message);
and
receive(source, &message);
```

- `send`: Envoyer un message vers une destination.
- `receive`: Recevoir un message d'une destination.
- Le récepteur peut bloquer jusqu'à l'arrivé d'un message.

Echange de messages: Pseudo-code

```
#define N 100 /* Nombre d'entrées du buffer */

void producer(void)
{
  int item;
  message m;
  while (TRUE) {
     item = produce item( ); /* generer msg */
     receive(consumer, &m); /* attendre
liberation d'espace */
     build message(&m, item); /* construire msg
a envoyer */
     send(consumer, &m); /* envoyer msg*/
}
}
```

```
void consumer(void)
{
  int item, i;
  message m;
  for (i = 0; i < N; i++) send(producer, &m); /*
envoyer places libres */
  while (TRUE) {
     receive(producer, &m); /* recevoir msg */
     item = extract item(&m); /* extraire msg */
     send(producer, &m);
     consume item(item);
  }
}
```

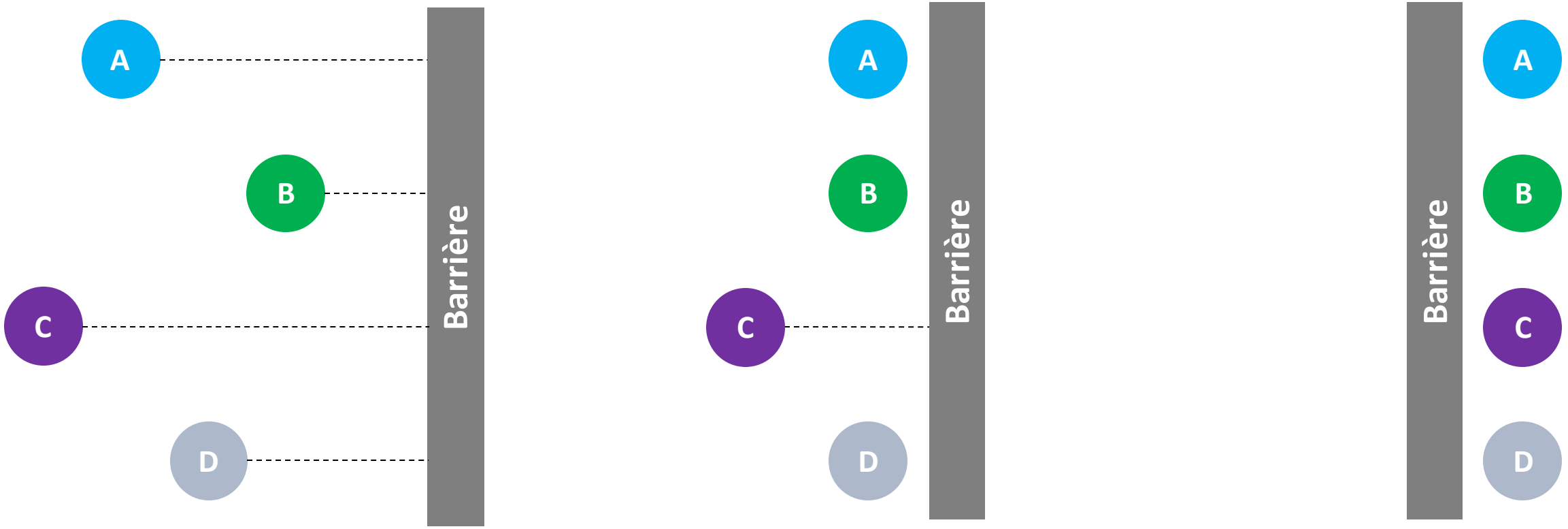
Echange de messages: problèmes

- Les messages peuvent être perdu si les machines sont connectés en réseau.
- Comment identifier les processus.
- L'authentification est un autre problème. Comment s'assurer qu'un client communique avec le vrai serveur.
- Performance.

Barrières (*Barriers*)

- Mécanisme de synchronisation pour un groupe de processus.
- Quand un processus finisse son calcul nécessaire durant une phase, il exécute la **primitive de barrière**.
- La prochaine phase ne commence que si tous les processus finissent la phase précédente.

Barrières: Exemple

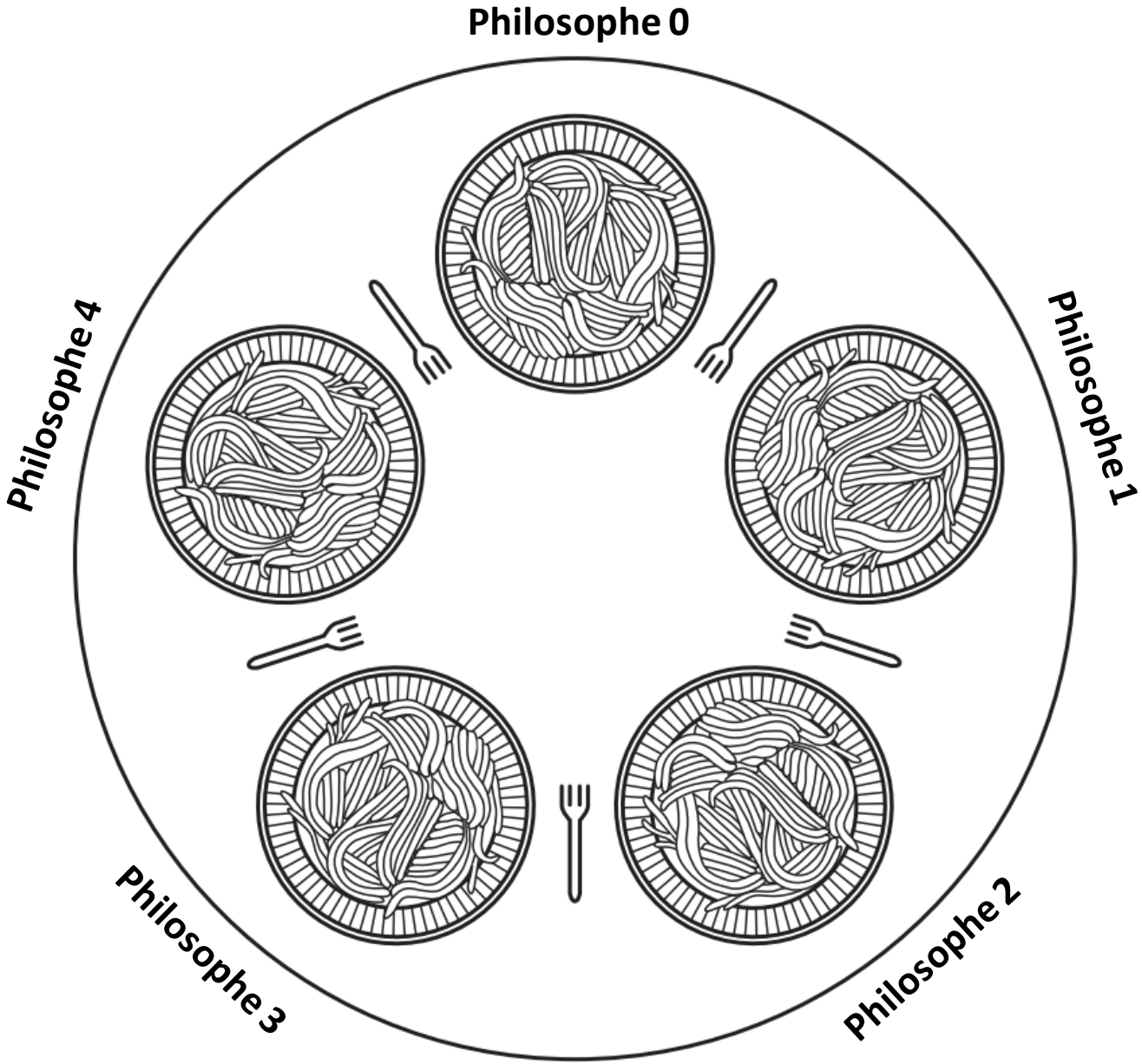


Autres problèmes classiques de communication inter-processus

Le problème des philosophes

- Introduit par *Djikstra* en 1965.
- Un ensemble de philosophes assis autour d'une table ronde.
- Chaque philosophe a un plat devant lui.
- Pour manger, un philosophe a besoin de deux fourchettes.
- Un philosophe alterne entre les périodes ou il mange et les périodes ou il pense.

Le problème des philosophes



Problème des philosophes: Première solution

```
#define N 5 /* number of philosophers */

void philosopher(int i) {
  while (TRUE) {
    think( );
    take fork(i);
    take fork((i+1) % N);
    eat( );
    put fork(i);
    put fork((i+1) % N);
  }
}
```

Penser

Prendre les deux fourchettes

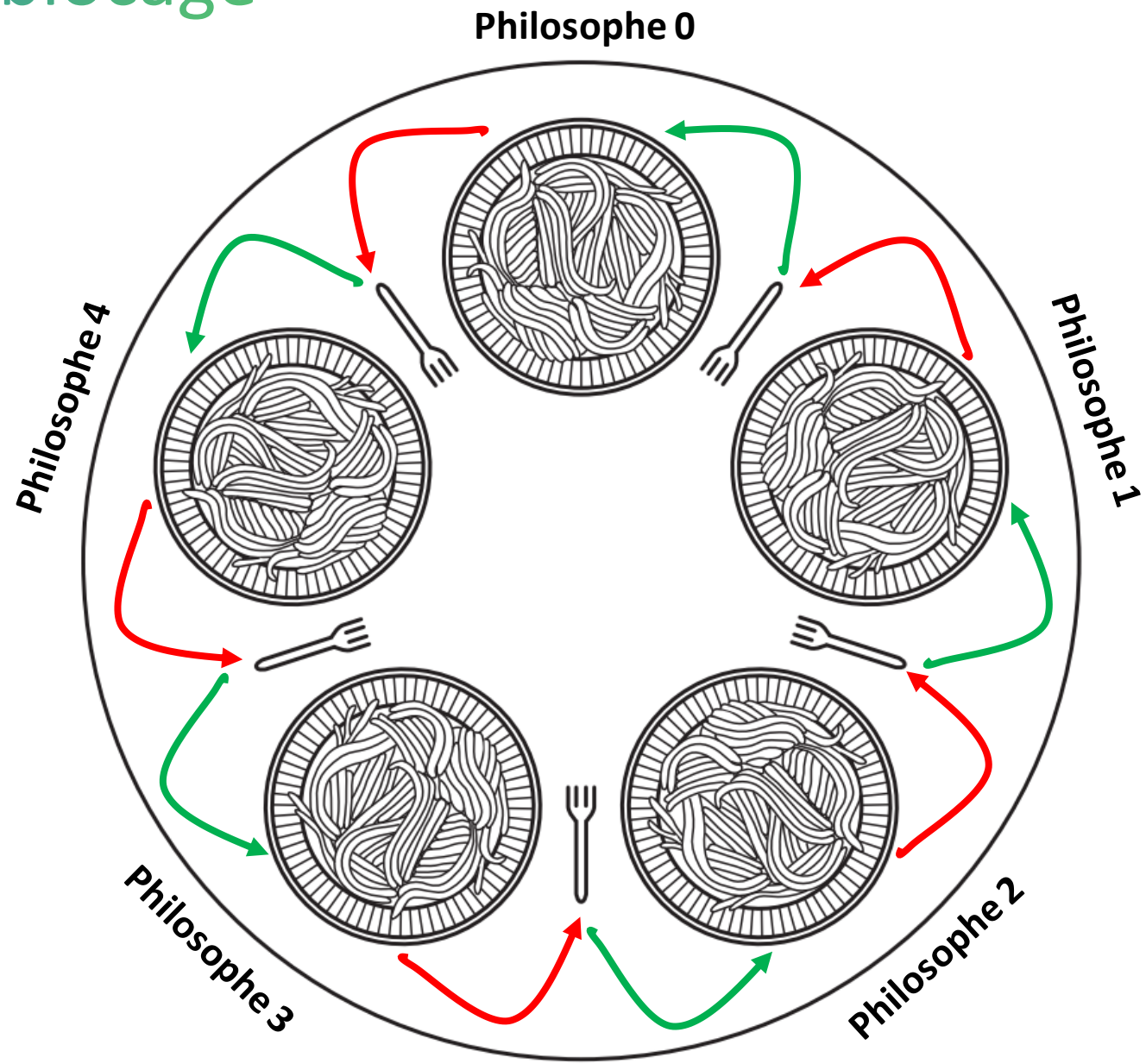
Manger

Remettre les deux fourchettes

Scénario

- Supposant que tous les philosophe prennent leur fourchette de gauche simultanément.
- Aucune fourchette de droite n’est disponible.
- **Interblocage**.

Interblocage



Amélioration à la solution

- Quand un philosophe saisie la fourchette de gauche:
- Il vérifie si la fourche de droite est libre.
- Si elle n’est pas libre, il remet la fourchette de gauche et attend un moment avant de recommencer.

Scénario

- Supposant que les philosophe saisissent leur fourchette en même temps.
- Vérifie que les fourchettes de droite sont occupées.
- Il remettent leur fourchette, attendent un moment puis démarre simultanément.

- La situation ou les programmes en exécution ne font aucun progrès est appelée: **starvation (*famine*)**.

Amélioration à la solution

- Attendre un moment aléatoire avant de recommencer.
- N’est pas fiable pour des applications critiques (station nucléaire).

Une autre amélioration

- En utilisant des **sémaphores**.

```
void philosopher(int i) {
  while (TRUE) {
     think( );
     P(mutex)
     take fork(i);
     take fork((i+1) % N);
     eat( );
     put fork(i);
     put fork((i+1) % N);
     V(mutex)
  }
}
```

Problème

- Un problème de performance.
- Un seul philosophe peut manger à un instant donné.
- On devrait permettre a deux philosophes de manger simultanément.

Solution efficace

- Elle utilise un vecteur d'etats (philosophes).
- Trois etats: ***`eating`***, ***`thinking`***, ***`hungry`***.
- Un philosophe peut être en état *eating* seulement si ses deux voisins ne sont pas entrain de manger.
- Un vecteur de sémaphores: un pour chaque philosophe.

Solution efficace: Initialisation

```
#define N 5                     /* Nombre de philosophes */
#define LEFT (i+N-1)%N          /* ID du voisin gauche */
#define RIGHT (i+1)%N           /* ID du voisin droit */
#define THINKING 0              /* philosophe entrain de penser */
#define HUNGRY 1                /* philosophe veut avoir les fourchettes */
#define EATING 2                /* philosophe entrain de manger */

typedef int semaphore;
int state[N];                   /* vecteur d'etats */
semaphore mutex = 1;            /* exclusion mutuelle */
semaphore s[N];                 /* vecteur de semaphores pour les philosophes */
```

Solution efficace

```
void philosopher(int i)
{
  while (TRUE) {
  think( );
  take forks(i);
  eat( );
  put forks(i);
  }
}

void test(i) {
 if (state[i] == HUNGRY && state[LEFT] !=
EATING && state[RIGHT] != EATING) {
 state[i] = EATING;
 up(&s[i]);
 }
}
```

```
void take_forks(int i) {
  down(&mutex);
  state[i] = HUNGRY;
  test(i);      /* try to acquire 2 forks */
  up(&mutex);   /* exit critical region */
  down(&s[i]); /* block if forks were not
acquired */
}

void put_forks(i) {
  down(&mutex); /* enter critical region */
  state[i] = THINKING;
  test(LEFT);   /* see if left neighbor can
now eat */
  test(RIGHT);  /* see if right neighbor
can now eat */
  up(&mutex);   /* exit critical region */
}
```

Problème des lecteurs/rédacteurs

- Introduit en 1971 (courtois et al.)
- Modélise l'accès a une base de données.
- Autoriser l'accès simultané a plusieurs lecteurs.
- Si un processus est entrain d'écrire, bloquer l'accès a tous les processus, même les lecteurs.

Lecteurs/rédacteurs: solution

```
typedef int semaphore;
semaphore mutex = 1;          /* controller l'accès à la section critique */
semaphore db = 1;             /* controller l'accès à la base de données */
int rc = 0;                   /* # de processus rédacteurs */
```

Problème du lecteurs/rédacteurs

```
void reader(void)
{
 while (TRUE) {
  down(&mutex);
  rc = rc + 1; /*incrementer # rédacteurs*/
  if (rc == 1) down(&db);
  up(&mutex);
  read data base( );
  down(&mutex);
  rc = rc – 1;
  if (rc == 0) up(&db);
  up(&mutex);
  use data read( ); /*region non critique*/
 }
}
```

```
void writer(void)
{
  while (TRUE) {
   think up data( ); /*region non critique*/
   down(&db);
   write data base( );
   up(&db);
  }
}
```