Chap. III SYNCHRONISATION ET COMMUNICATION ENTRE PROCESSUS

(et threads)

(IPC – Inter Process Communication)

Plan des IPCs

1) Outils de **synchronisation** à attente active (TSL, verrous)

2) Sémaphores

3) Moniteurs

4) Problèmes classiques de synchronisation: RDV, P/C, L/R, philosophes, …

5) Les IPCs threads (Posix + java en complément)

   - Mutex, sémaphores, variables conditionnelles

6) Communication –par passage de messages

   a) Les pipes Unix

   b) Les signaux Unix

Exemple introductif

Int i;

P1

```
i=0;
while (i<10)
    i++;
printf("P1 GAGNE! \n");
```

P2

```
i=0;
while (i>-10)
    i--;
printf("P2 GAGNE !\n");
```

- ✓ i variable partagée → risque de conflit d'accès!
- ✓ Les instructions i++ et i– doivent s'exécuter de manière indivisible!
- ✓ Lequel des processus P1 ou P2 gagne?
- ✓ Vont-ils terminer? Si l'un se termine, est-ce que l'autre termine aussi?
- ✓ Est-ce que P1 peut commencer?

Présentation du problème

Pas d'interaction:

- Exécution dans n'importe quel ordre
- Exécution parallèle ou concurrente

✓ ***Interactions entre processus:***

- Nécessité de synchroniser
  - L'ordre d'exécution est important
  - Cas particulier: Exclusion mutuelle –sérialisation des exécutions

✓ ***Moyens de synchronisation:***

- Matériel: masquage d'interruption et TAS—Test-And-Set
- Logiciel: verrous, sémaphores, moniteurs, et passage par messages

Niveau d'abstraction → (- à +)

Définitions de synchronisation, Section critique

- **Synchronisation**
  - Utilisation d'opérations atomiques afin de garantir une bonne coopération entre processus.
  - Une opération qui consiste à distribuer, dans le temps, les accès à une ressource partagée entre plusieurs processus.
- **Conditions de rapidité**
  - un ordre quelconque de processus/instructions peut produire des résultats incorrects
  - La commutation dépend, dans le cas
    - de concurrence, de l'ordonnancement
    - de parallélisme, de la vitesse d'exécution relative– on synchronise le processus le plus rapide sur le processus le plus lent.

Section critique --SC

- ✓ Une partie d'un programme où se produit un conflit d'accès.
- ✓ Comment éviter ce conflit?
  - ➢ Besoin de contrôler l'entrée à une SC
  - ➢ Besoin de supporter l'exclusion mutuelle dans la SC.
- ✓ Une bonne solution au problème de SC doit satisfaire:
  1. ***Exclusion mutuelle (***accès exclusif): à tout instant un seul processus exécute sa SC (ressource partagée).
  2. ***Avancement et absence de blocage***: un processus qui n'est pas dans sa SC ne doit bloquer un autre processus à entrer en SC; c-à-d pas d'attente s'il n'y a pas de compétition
  3. ***Attente bornée (pas de famine***): une fois la demande d'entrée en SC est lancée, le processus ne doit pas attendre indéfiniment. La demande est assurée de manière équitable, si possible.

⇨ Aucune hypothèse ne doit être faite sur les vitesses relatives des processus

Structure Typique d'un Processus

Soient N processus exécutant le programme suivant:

Do

…

// ”Entrer en SC” } **Prologue**

SC

// ”Sortir de la SC” } **Epilogue**

SNC

While (1);

Solutions Possibles

- ✓ *Hypothèses:*
  - ➢ Vitesses relatives des processus quelconque et inconnu
  - ➢ tout processus quitte sa SC au bout d'un temps fini
- ✓ Aperçu

Opérations atomiques de haut niveau (API)	*Verrous* *Sémaphores* *Moniteurs* *Send / Receive* → (- +)
Opérations atomiques de bas niveau (matériel)	*Load / Store* *Masquage Interruption* *TAS*

Solutions Matérielles

- ✓ Permettre à l'utilisateur d'interdire momentanément les interruptions (difficile et dangereuse!)
- ✓ Augmenter l'ensemble des actions atomiques
- ✓ *Masquage d'interruptions*
  - ➢ Problème: les processus users ne peuvent pas garantir le test et la modification d'une variable
  - ➢ Solution: Interdire la commutation de processus pendant qu'un processus est en SC ou encore masquer les Its (le système peut le faire en mode SVC par un appel spécifique).

Masquer It

SC

Demasquer It

Solutions Matérielles –TAS (TSL)

- TAS --Test-And-Set (ou TSL –Test and Set Lock): Instruction spéciale câblée dont le rôle est de rendre atomique le ''test and set'' du contenu d'un mot.

```
Int TAS (int *val)
{ int temp;
  temp= *val;                    //implantée de manière atomique
  *val = 1;
  return temp;
}
```

- Une solution au problème de SC pour n processus:

```
Int verrou = 0;
Processus Pi
Do
        MutexDebut();
        SC
        MutexFin();
        SNC
while (1);
```

```
void MutexDebut()
{
    while (TAS(&verrou))
            ;
}
void MutexFin()
{   verrou = 0; }
```

- Preuve?

Solutions Logicielles –Attente Active sur un Verrou

- Soient deux processus P0 et P1
- **Algorithme1: A qui le tour!**

```
Public    //Variables partagées

    int tour=0; //tour =i si Pi veut entrer en SC
Processus Pi
    Do
            while (tour != i)
                        ;           // on fait rien
            SC
            tour = (i+1)%2;
            SNC
      While (1);
```

- Démontrer que c'est une fausse solution?
  - Exclusion mutuelles satisfaite
  - Avancement non vérifié: si P0 est plus lent que P1 alors P0 bloque P1; bien qu'il n'est pas dans sa SC
  - Nécessité d'une alternance stricte (jeton)

Algorithme2: deux drapeaux

```
Public    //Variables partagées
    int flag[2]=0[2];  //flag[i]=0 si Pi est prêt pour entrer en SC
```

Processus Pi

```
    Do
            flag[i] = 1;
            while (flag[j])
                ;          // on fait rien
            SC
            flag[i] = 0;
            SNC
    While (1);
```

- ✓ Démontrer que c'est une fausse solution?
  - ➢ Exclusion mutuelles satisfaite
  - ➢ Avancement non vérifié: si les processus arrivent en même temps, c.-à-d. flag[0] = flag[1] =1!
- ✓ Une vraie solution consiste à combiner les deux dernières!

Algorithme3: Solution de Peterson (2 processus)

```
Public    //Variables partagées
    int flag[2]=[0];  //flag[i]=0 si Pi est prêt pour entrer en SC
    int tour = 0;
```

Processus Pi

```
    Do
            flag[i] = 1;
            tour = j;                          // j = (i+1)%2;
            while (flag[j] && tour == j)
                ;        // on fait rien
            SC
            flag[i] = 0;
            SNC
    While (1);
```

- ***Preuve de correction?***
  - Cet algorithme satisfait les 3 conditions de SC, à démontrer?
- Généralisation à n processus: voire algorithme de Bakery

Attente active --Conclusion

- Solutions qui fournissent des attentes actives
  - Inefficace: il faut que le processus en attente libère le processeur explicitement (exemple la fonction sleep sous Unix).
  - Les processus de priorité élevé peuvent être prives (inversion de priorité)
- Solutions de blocage
  - Sémaphores
  - Moniteurs
  - Send/Receive

Les SEMAPHORES

✓ ***Motivation : synchronisation des processus concurrents***

- Une approche par *attente active* n'est pas intéressante, puisque le processeur est immobilisé simplement pour attendre
  - Gaspillage de la puissance CPU disponible
  - Une approche alternative = utilisation de sémaphores

✓ ***Principe et définition des sémaphores***

- Mécanisme de synchronisation simple et ancien entre des processus concurrents
- Le principe est directement hérité des chemins de fer -- Signal muni d'un bras indiquant si la voie ferrée est libre ou occupée
  - Sémaphore levé : le processus P peut continuer son chemin
  - Sémaphore baissé : il doit attendre jusqu'à ce qu'un autre processus Q le lève
  - Eviter des collisions en assurant l'accès exclusif à un croisement ferré

✓ **Syntaxe et Sémantique**

- Un sémaphore S est une structure de données manipulant uniquement 3 opérations atomiques : initialisation, P et V sur une variable entière.

Sémaphores: inventé par Dijkstra

Edsger Dijkstra

- ✓ **Né à Rotterdam le 11 mai 1930 et mort à Nuenen le 6 août 2002, est un mathématicien et informaticien néerlandais du XXe siècle**

Syntaxe et Sémantique d'un Sémaphore

- ***P --passer : P(S) / Wait(S) / Down(S)***
  - Décrémenter la variable S (à moins qu'elle ne soit déjà à 0)
  - Utilisée (lorsque déjà 0) pour bloquer (suspendre) le processus appelant jusqu'à ce qu'un événement survienne.
- ***V --relâcher : V(S) / Signal(S) / Up(S)***
  - Incrémenter le sémaphore de 1
  - Utilisée pour signaler un événement, et si possible, réactiver un processus en attente.
- ***Initialisation de S:*** interprétée comme un nombre d'autorisations (disponibles quand l'entier est positif, attendues quand le nombre est négatif).
- ***Déclaration de sémaphores -- Notation d'Andrews***
  - Sem S1, S2; Sem ingred[3]=([3] 1); S1 = 0; S2 = 1;
  - Apres initialisation, les seules opérations permises sont P et V
- ***Sémaphores général vs. sémaphore binaire :***
  - Sémaphore général : peut prendre n 'importe quelle valeur non-négative
  - Sémaphore binaire : la valeur peut être uniquement 0 ou 1

Syntaxe et Sémantique d'un Sémaphore (suite)

- ***Sémaphore associé à une file d'attente -- Sémaphore de blocage***
  - A chaque sémaphore est associée une ***file d'attente pour les processus bloqués***
- **Création d'un sémaphore général**

```
typedef struct semaphore {
        int valeur;
        bcp *tete; }

void P(semaphore *S)
   { if ( --S->valeur < 0) {
            insérer ce processus dans la FA associée
            se Bloquer }
   }

void V(semaphore *S)
   {  if ( ++S->valeur <= 0) {
            supprimer un processus courant de la FA associée à ce sémaphore
            Réveiller  un processus (→prêt)}
   }
```

- L'utilisation correcte des sémaphores ne doit pas dépendre d'une gestion particulière de la file d'attente

Utilisation des Sémaphores

- Les sémaphores peuvent être utilisés tant pour la réalisation des *sections critiques* que pour diverses formes de *synchronisation conditionnelle*

Compétition entre 2 processus

Variables partagées

Semaphore Mutex=1;

Processus Pi

```
Repeat
  P(Mutex);
   SECTION CRITIQUE
  V(Mutex);
until flase;
```

Coopération -- sémaphore privé

Variables partagées

Semaphore Sync = 0;

Processus P0	***Processus P1***
…….	…….
P(Sync);	*V(Sync);*
…….	…….

Il existe une relation de précédence P1 < P0

- ***Conséquence :***
  - Un sémaphore est toujours initialisé à une valeur non-négative mais peut devenir négative après un certain nombre d'opérations P(S) -- nombre des processus en attente.

Problèmes de dysfonctionnement des sémaphores

✓ ***Remarques :***

- La plupart des mises en œuvre des sémaphores assurent que les processus en attente sont réactivés dans l'ordre dans lequel ils ont été suspendus sur le sémaphore.
  - Equité dans l'ordonnancement des processus
- Les sémaphores sont les principales primitives de synchronisation dans Unix
- Un sémaphore est un mécanisme qui permet le blocage et le réveil explicite.
  - Servir à traiter tous les paradigmes de la programmation concurrente

☠ ***Aucune garantie qu'une synchronisation est exempte de problèmes!! :***

- ***Interblocage*** (Deadlock) -- attente circulaire
  - Un processus est bloqué indéfiniment s'il est en attente d'un événement qui ne peut être produit que par le processus déjà en attente
  - Considérons 2 processus utilisant 2 sémaphores d'exclusion mutuelle

P1	*P2*
P(S1)	*P(S2)*
P(S2)	*P(S1)*
…	…

- ***Famine*** (starvation) : des processus qui s'exécutent indéfiniment sans aucun changement; certains processus peuvent ne jamais obtenir les ressources!

Attente active vs. Blocage

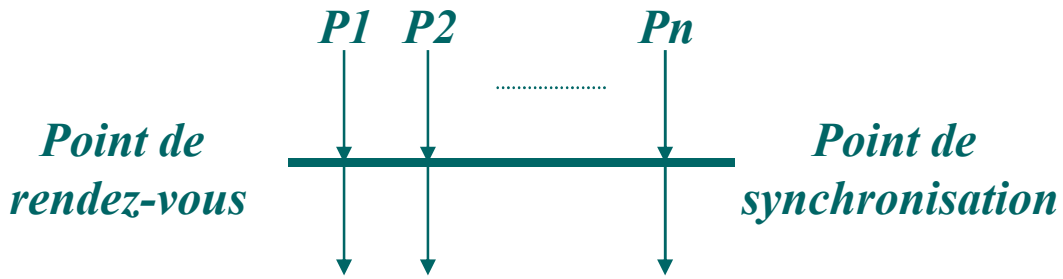
- ✓ L 'attente active est-elle plus coûteuse que le blocage?
- ✓ Coût de blocage contre le coût de manipulation des files d 'attente + la commutation de contexte?
- ✓ La durée de l 'attente?
  - ➢ L'attente active peut être meilleure pour des sections critiques de courtes durées, plus particulièrement pour les multiprocesseurs, elle est incontournable.

Problèmes Classiques de Synchronisations

- ✓ Problème du Rendez-vous (RDV) et/ou Barriere
- ✓ Problème des Producteur(s)-Consommateur(s)
- ✓ Problème des Lecteurs- Rédacteurs
- ✓ Problème du diner des philosophes (spaghettis avec 2 fourchettes)

Problèmes Classiques de Synchronisations

- ***Rendez-vous -- Principe général***



- Version simplifiée du problème pour 2 processus (généralisation -- voir TD)
  - Synchronisation par sémaphores privés :
  - Semaphores arrivee1=0, arrivee2 = 0;

Processus P1

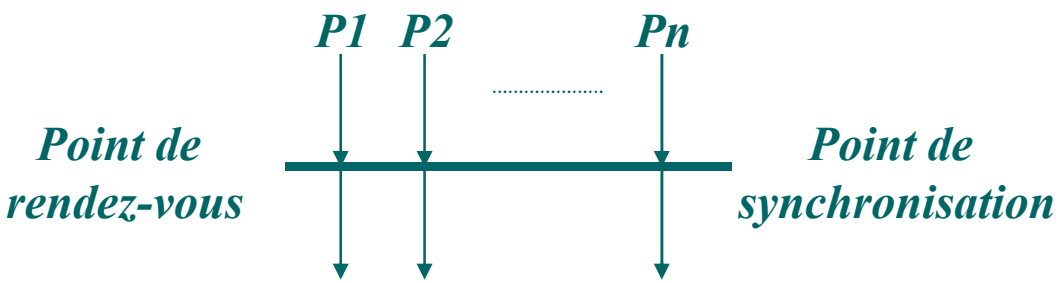
```
……
V(arrivee1);      // signaler mon arrivée
P(arrivee2);      // attendre l'arrivée de l'autre
……
```

Processus P2

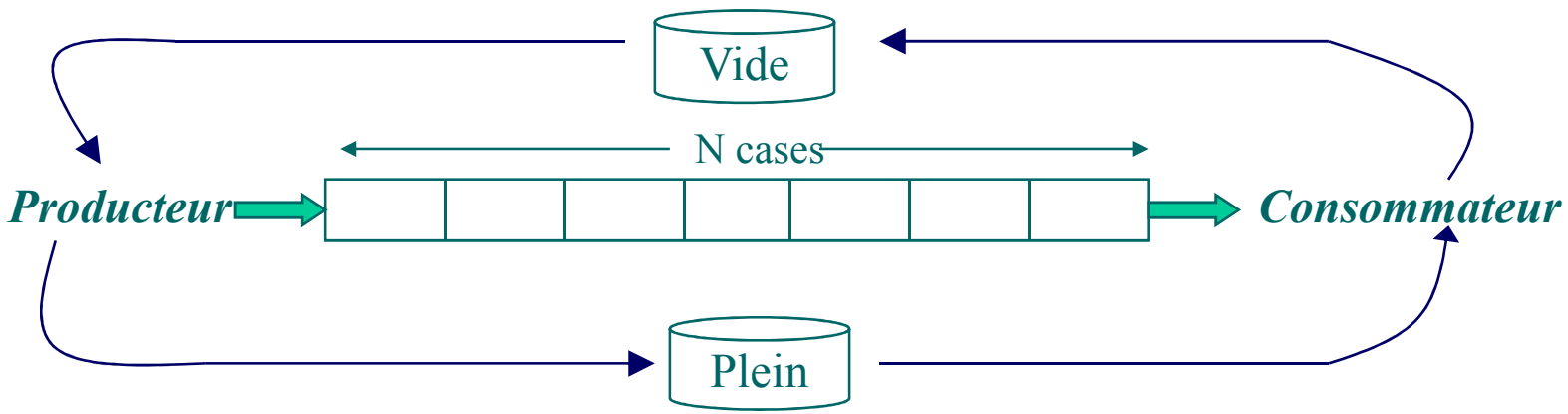
```
…….
V(arrivee2);
P(arrivee1);
……
```

TD (en classe)

- ✓ *Généraliser la solution du problème de RDV pour n processus*



Problème de Producteur/Consommateur (tampon borné)



- ***Contraintes de synchronisation :***
  - Relation de précédence : Producteur < Consommateur
  - Section critique (tampon)
    - tampon plein ⇨ Producteur se bloque
    - tampon vide ⇨ Consommateur se bloque
    - Exclusion mutuelle au tampon

Solution au Problème de Producteur/Consommateur (tampon borné)

✓ ***Variables partagées***

```
#define N 100
Semaphore mutex=1;  /* protège l'accès au tampon */
Semaphore plein=0;  /* compte le nombre d 'informations produites dans le tampon */
Semaphore vide = N;  /* Nb d 'emplacements libres dans le tampon */
```

Processus Producteur	***Processus Consommateur***

Repeat
……
Produire_objet();
……
P(vide) /* dec. Cases libres */
P(mutex); /* entrée en SC */
deposer_objet();
V(mutex); /* sortie de SC */
V(plein); /* Incr. nb info. */
until false;
	Repeat
	……
	P(plein); /* décrémenter nb info. */
	P(mutex); /* entrée en SC */
	retirer_objet();
	V(mutex); /* sortie de SC */
	V(vide); /* Incr. nb cases vides */
	Consommer_objet()
	until false;

Problème des Lecteurs/Rédacteurs

- Considérons ce problème comme étant un système de réservation de billets d'avions où plusieurs processus tentent de lire et d'écrire des informations:
  - On accepte que plusieurs lisent ensemble (degré d'accès >= 1)
  - On n'autorise qu'un seul processus à modifier (on exclut les lecteurs et les autres rédacteurs) ➔ Exclusion mutuelle (degré d'accès = 1)
  - On suppose que les lecteurs sont prioritaires par rapport aux rédacteurs
  - Un rédacteur bloqué doit attendre le dernier des lecteurs pour qu'il puisse entrer en section critique
- **Solution**

Variables partagées

```
Semaphore mutex1=1;      /* protège le compteur des lecteurs */
Semaphore mutex2=1;      /* garantir la priorité des lecteurs)
Semaphore wrt=1;         /* exclusion mutuelle pour les rédacteurs */
int nblect=0;            /*  Nombre de lecteurs arrivés */
```

Solution au Problème des Lecteurs/Rédacteurs avec priorité des lecteurs par rapport aux rédacteurs

Processus Lecteur	Processus Rédacteur
.....	
P(mutex1); /* accès exclusif à nblect */	P(mutex2); /* priorité des lecteurs*/
if (++nblect == 1)	P(wrt); /* accès exclusif */
P(wrt); /* se bloquer ou causer le blocage les rédacteurs */	**Ecriture**
V(mutex1); /* libérer l'utilisation de nblect */	V(wrt); /* libérer l'accès exclusif */
Lecture	V(mutex2);
P(mutex1);	
if (--nblect == 0) /* si le dernier lecteur */	
V(wrt); /* autoriser une écriture */	
V(mutex1);	

TD (en classe)

Solution au Problème des Lecteurs/Rédacteurs

avec priorité des rédacteurs par rapport aux lecteurs

Les MONITEURS

✓ ***Motivation :***

⇨ Les sémaphores peuvent être utilisés pour résoudre à peu près n'importe quel problèmes d'exclusion mutuelle ou synchronisation … mais les sémaphores possèdent certains désavantages :

- Mécanisme de bas niveau qui demande une discipline sévère dans la façon dont ils sont utilisés, sous peine d'erreurs: que se passe-t-il si on oublie d'indiquer un appel à V? ou si on effectue une action P en trop?
- Le rôle d'une opération P ou V (exclusion mutuelle? synchronisation conditionnelle?) dépend du type de sémaphore, de la façon dont il est initialisé et manipulé par les divers processus pas explicite

✓ ***Moniteur :***

⇨ Mécanisme de synchronisation de haut niveau, proposé par Hoare et Brinch Hansen.

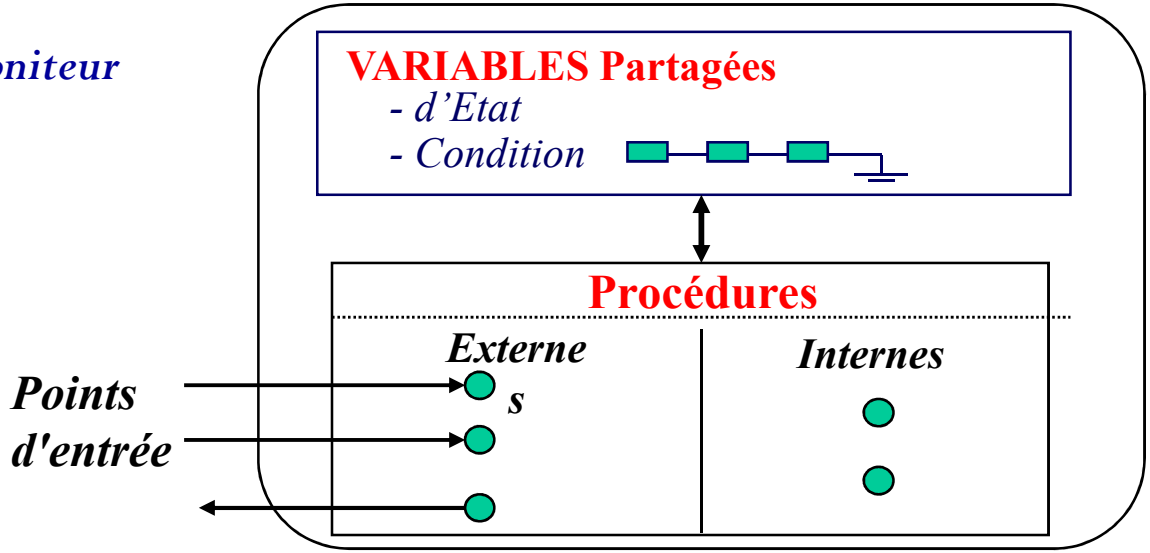
⇨ Forme de module qui supporte, à l'aide de deux mécanismes indépendants, l 'exclusion mutuelle et la **synchronisation conditionnelle**.

⇨ Conceptuellement, un moniteur simule une classe en OO (des variables partagées et les méthodes qui les manipulent)

⇨ Un moniteur est censé assurer une exclusion mutuelle (un seul processus actif dans le moniteur) d'accès aux données qu'il contient

Syntaxe et Sémantique d'un Moniteur

✓ *Structure d'un Moniteur*



✓ ***Sémantique d'un Moniteur =***

- *Type abstrait*, mais avec des propriétés d'exclusion mutuelle et de synchronisation lorsque le moniteur est partagé par plusieurs processus.
  - On n'a accès qu'aux procédures externes, pas aux variables
  - Les procédures sont exécutées en exclusion mutuelle et donc les variables internes sont manipulées en exclusion mutuelle
  - On peut ***se bloquer*** et ***réveiller d'autres processus***. Le blocage et le réveil s'exprime au moyen de *conditions*.

Syntaxe et Sémantique d'un Moniteur (suite)

✓ ***Syntaxe :*** la forme générale d'une déclaration de moniteur :

```
Monitor nom_moniteur {
        /* ---  Déclarations des variables  --- */
         .....;  /* variables d'états */
        Condition ... ; /* variables  conditions */
       /* ------------ Déclarations des procédures    ------------ */
        Public nom_fonction (...)
         {          ....       }
        Public void nom_procedure (..)
        {           ....                          }
        Private .... (...)
        {       .....        }
        {  /* -- Initialisation des variables ---*/
        }
```

Exclusion Mutuelle

- **Philosophie des moniteurs** = séparer de façon claire l 'exclusion mutuelle de la synchronisation conditionnelle (coopération):
  - L'exclusion mutuelle est supportée de façon ***implicite*** : un appel, par un processus, d'une procédure exportée par le moniteur assure que la procédure sera exécutée de façon ***exclusive***, c-à-d, au plus un appel d'une procédure du moniteur sera actif à un instant donné → le moniteur maintient une FA des processus en attente d'entrée.
  - Les synchronisations conditionnelles doivent être décrites de façon explicite à l 'aide de variables condition (Condition variables)
- En d'autres termes, ***l'exclusion mutuelle est automatique***, sa mise en œuvre étant assurée par le langage (compilateur), la librairie, ou le système d 'exploitation, pas le programmeur lui-même.
- ***Langages de programmation***
  - JAVA (le meilleur) déclarations en exclusion mutuelle (''synchronized'') certaines méthodes d 'une classe.
  - ADA95 -- type protégé, objet protégé « protected »; ainsi, toutes les procédures de ces objets protégés sont exécutées en exclusion mutuelle.

Variables de Condition

- Une variable condition est utilisée pour suspendre un processus jusqu'à ce qu'une certaine condition devienne vraie.
  - Déclarée comme une variable, mais on ne peut ni lui attribuer de valeur ni la tester) -- Condition C;
  - Servir de FA des processus qui attendent sur cette condition
  - 3 opérations possibles sur une variable condition C :
    - ***Empty(C)*** // La FA associée est-elle vides ?
    - Mise en attente d'un processus : ***Wait(C)*** // le processus appelant se bloque et doit libérer le moniteur.
    - Réactivation de processus en attente : ***Signal(C)*** // reprend exactement un processus (en tête de la FA associée à C). Si aucun processus n 'est suspendu alors cette opération est sans effet.
- **Problème de signalisation :** un processus P fait un signal et réveille un processus Q, alors qui aura le contrôle exclusif du moniteur? 2 approches :
  - *Signaler et continuer* : le processus qui exécute signal continue son exécution, donc conserve l'accès exclusif au moniteur. Le processus ayant été signalé sera exécuté plus tard → *Approche non-préemptive -- la plus couramment utilisée (Unix, Java, Pthreads).*
  - *Signaler et Attendre* : le processus qui signale attend pendant que celui qui vient d'être signale acquiert l accès exclusif au moniteur

Similitudes/Différence entre P/Wait et V/Signal

- ✓ Les opérations *Wait* et *P* peuvent toutes deux avoir pour effet de suspendre un processus qui exécute cette opération :
  - ➢ *Wait* suspend toujours le processus
  - ➢ *P* ne le fait que si la valeur du sémaphore est négative ou nulle
- ✓ *Signal* et *V* peuvent réactiver un processus suspendu :
  - ➢ *Signal* n 'a aucun effet si aucun processus n'est suspendu,
  - ➢ alors que **V** aura pour effet d'incrémenter la valeur du sémaphore si aucun processus n 'est suspendu.
- ✓ ***Implantation des moniteurs par des sémaphores :***
  - ➢ Assurer l 'exclusion mutuelle au moniteur *mutex* (*P* entrée *V* après sortie)
  - ➢ A chaque variable condition sont associés un sémaphore et un compteur
  - ➢ Wait (V(mutex); P(semcond))

Exemples classiques de synchronisation -- Solutions par Moniteurs

- ✓ Les problèmes de RDV à n processus et des Lecteurs/Rédacteurs avec priorité des lecteurs (***en classe***).
- ✓ Problème de Producteur/Consommateur à tampon borné
- ✓ **Monitor** PC_TB {

```
#define N 100;

private objet Tampon[N];     //tampon borné

private int count=0;         //nombre d'objets déposés

private condition plein, vide;
```

Moniteur de Producteur/Consommateur

```
public void deposer (objet x)
{
    if (count == N)
        wait(plein);
    deposer_objet(tampon, x);
    count++;
    signal(vide);
}
}
```

```
public void retirer (objet x)
{
    if (count == 0)
        wait(vide);
    retirer_objet(tampon, x);
    count--;
    signal(plein);
}
```

Problèmes classiques avec Moniteurs
(à faire en classe)

1. *Problème du RDV (n)*

2. *Problème des L/R (avec différentes priorités)*

IPC (suite)
Synchronisation avec les threads

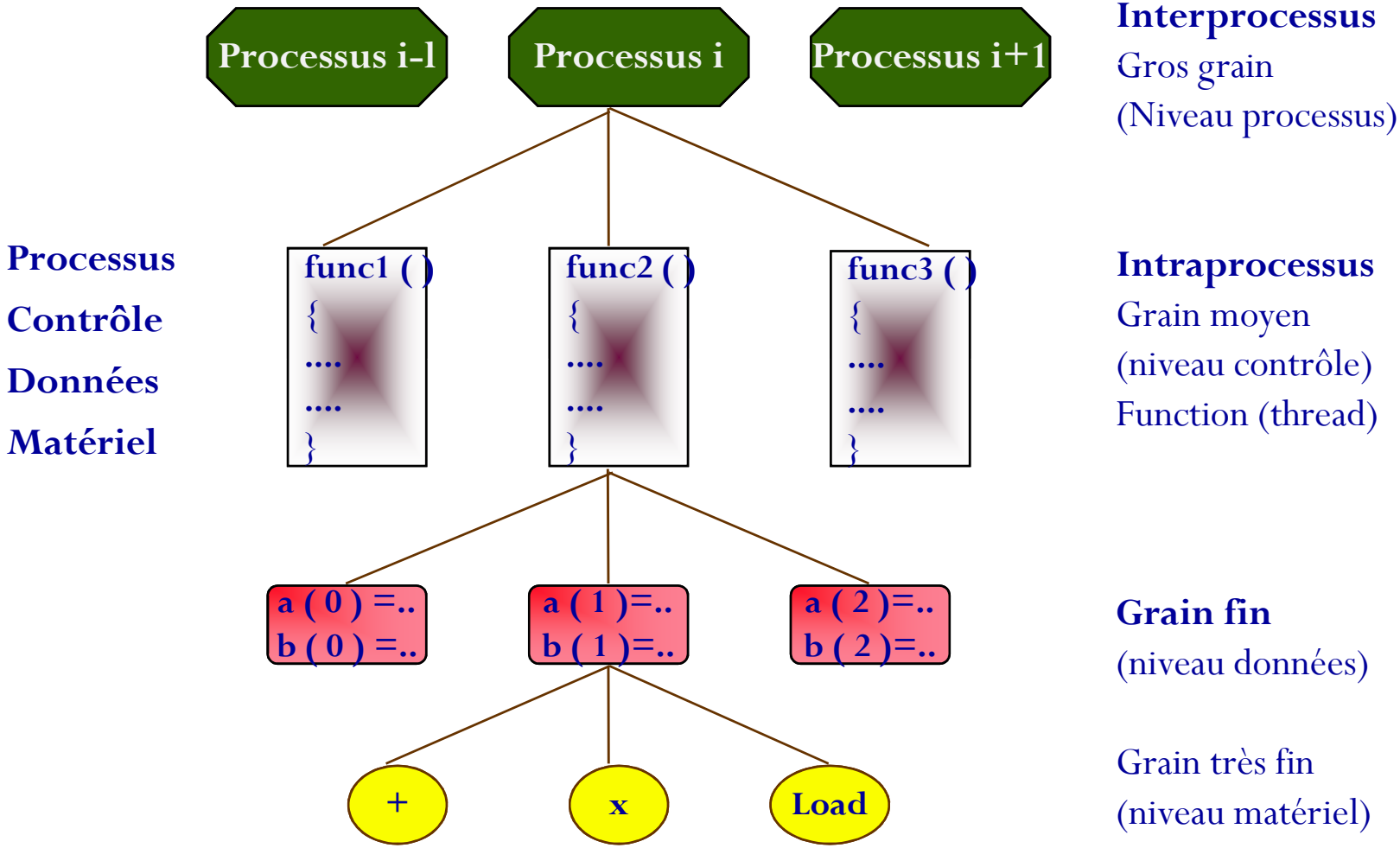
PLAN

① Introduction

② Primitives de synchronisation avec PThreads

- Verrous (mutex)
- Sémaphores
- Variables conditionnelles

Introduction --Niveau de Concurrence



Exemple Simple de Thread POSIX

```
void *func ( )
{
     /* define local data */
     - - - - - - - - - - - -
     - - - - - - - - - - - -        /* code de la function */
     pthread_exit(exit_value);
}

int main ( )
{
     pthread_t tid;
     int exit_value;
     - - - - - - - - - - - -
     pthread_create (&tid, NULL, func, NULL);
     - - - - - - - - - - - -
     pthread_join (tid, &exit_value);
     - - - - - - - - - - - -
     exit(0);
}
```

Introduction --Synchronisation des THREADS

- La plupart des programmes multithreadés ont des threads qui interagissent les uns avec les autres.
  - Partage de variables globales –Accès concurrent (R?/W?)
  - Besoin de synchroniser: imposer un ordre d'exécution des threads ou sérialiser l' accès à la ressource partagée

- **Nécessité de Protection**
  - Modification concurrente < x = 0x01> || < x = 0x100> x= ?
  - Exécution concurrente < a=x; x=a+ 1;> || < b=x; x=b -1;> x = -1/0/1?
  - Cohérence mémoire : < x = 1; y = 2;> || < printf("%d %d ", x, y); > affichage de 0/2?

Actions/Instruction Atomique? --Rappel

- Une action/instruction qui doit commencer et finir à s'exécuter sans aucune possibilité d'interruption
  - Une instruction machine a besoin d'être atomique (pas toutes!)
  - Une ligne de code C a besoin d'être atomique (pas toutes!)
  - a besoin d'être atomique (pas toutes!)
- Peut-on fournir une instruction atomique complexe?
  - Une section de code qui est forcée à être atomique est une **section Critique**

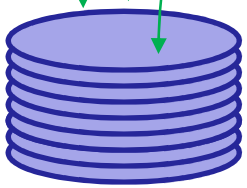
Section Critique: Mauvaise Programmation!

T1

```
reader()
{
  - - - - - - - - - -
  lock(DISK);
  .........
  .........
  .........
  unlock(DISK);
  - - - - - - - - -
}
```

T2

```
writer()
{
  - - - - - - - - - - -
  ..............
  ..............
  - - - - - - - - - - -
}
```



Shared Data

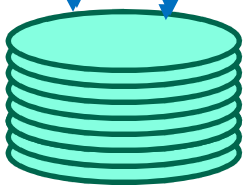
Section Critique: Bonne Programmation!

T1

```
reader()
{
...........
lock(DISK);
   ...........
   ...........
   ...........
unlock(DISK);
...........
}
```

T2

```
writer()
{
...........
lock(DISK);
   ..............
   ..............
unlock(DISK);
...........
}
```



Shared Data

Synchronisation Pthread

- **Plusieurs outils de synchronisation :**
  - les **verrous** -- locks : (pour assurer l'exclusion mutuelle--**MUTEX**)
    - Verrous lecteur/rédacteur : permet de réaliser une synchronisation de type lecteur/rédacteur
  - les **sémaphores** : pour résoudre l'exclusion mutuelle, pour synchroniser.
    - Problème : interblocage possible
  - les **variables conditionnelles** -- condition variables : pour attendre qu'un état, décrit par un prédicat soit vrai
    - elles résolvent le problème de l'interblocage.
  - les **barrières** : une barrière bloque un ensemble de threads jusqu'à ce que tous les threads aient atteint la barrière.

Synchronisation - Faux Exemple (du TP gestion des threads)

- Illustration du problème de partage de mémoire par les threads : quelle sera la valeur affichée par main ?

```
longint val = 0;
void main(void)
{   pthread_t tid1, tid2;
    void fonc(void);
    pthread_set_concurrency(2);
    pthread_create(&tid1, NULL, (void *(*)()) fonc, NULL);
    pthread_create(&tid2, NULL, (void *(*)()) fonc, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("valeur = %ld\n", val);
}
```

```
void fonc(void)
  {  long int i;
     pthread_t tid;
     for (i=0; i <1000000; i++)
          val  =  val + 1;
     tid = pthread_self();
     printf("tid : %d valeur = %d\n", tid, val);
     }
```

- Remarque : join est nécessaire. Sans join le processus n'attend pas la fin des threads, il se termine et entraîne la fin de tous les threads créés.

Synchronisation – Faux Exemple (suite)

- Voici les résultats de quelques exécutions différentes, pourquoi ces différences ?

  tid : 4 val = 1342484

  tid : 5 valeur = 1495077

  val = 1495077

  tid : 4 val = 1279685

  tid : 5 val = 1596260

  val = 1596260

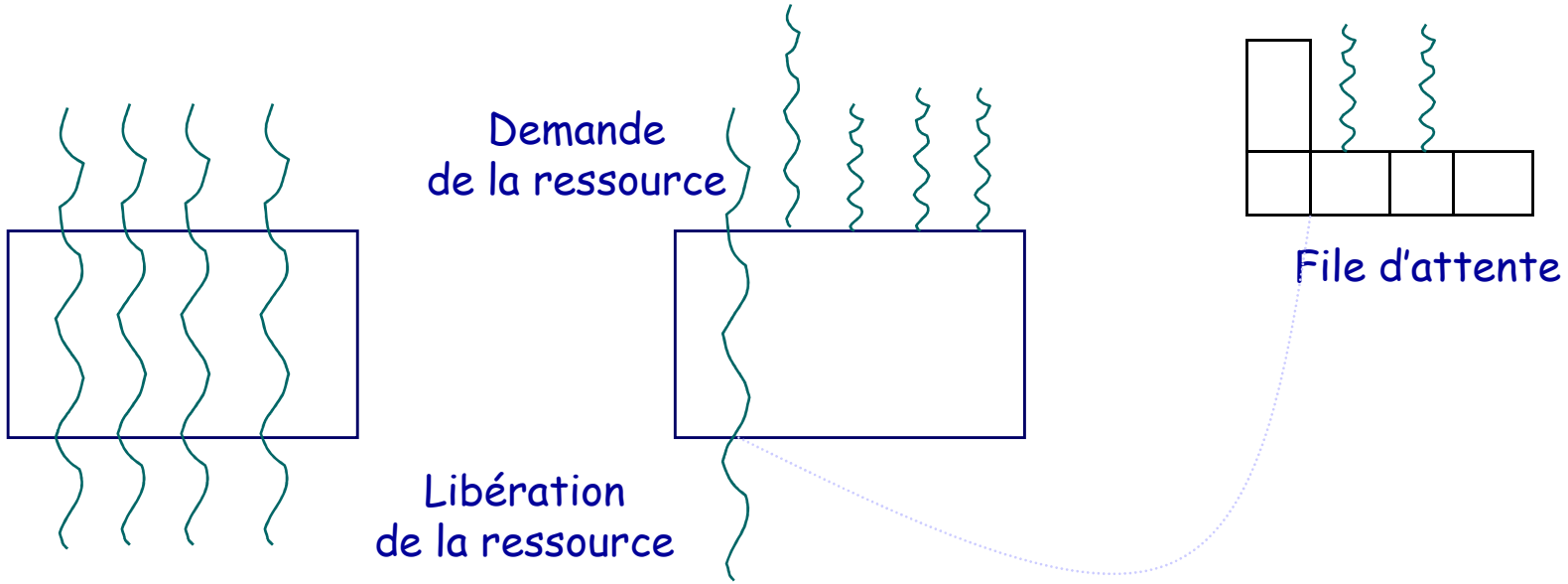
  tid : 5 val = 1958900

  tid : 4 val = 2000000

  val = 2000000

- L'incrémentation de valeur se fait en plusieurs instructions machine!:
  - Ranger valeur dans un registre
  - Incrémenter le registre
  - Ranger le registre dans valeur
- Si la fin de quantum intervient après la phase 1, le contexte mémorise l'état du registre à cet instant. Lors de la restauration du contexte, le registre reprend cette valeur, les incréments effectués entre-temps par l'autre thread seront donc écrasés…

Synchronisation -- Verrou d'Exclusion Mutuelle



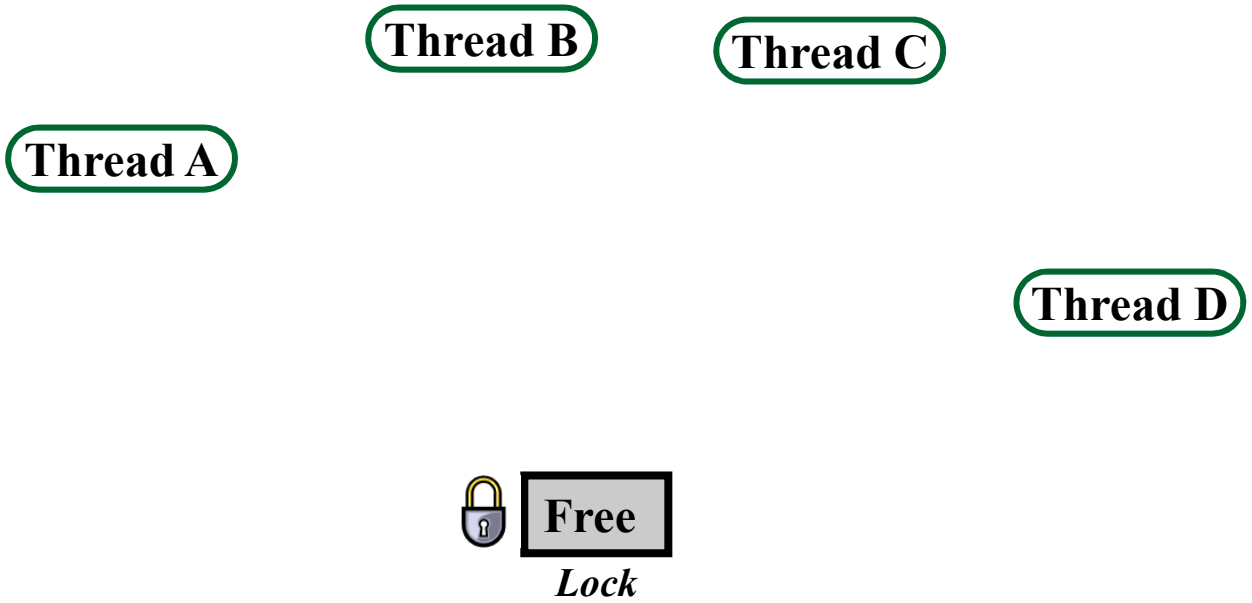
Section critique

Compétition pour une ressource

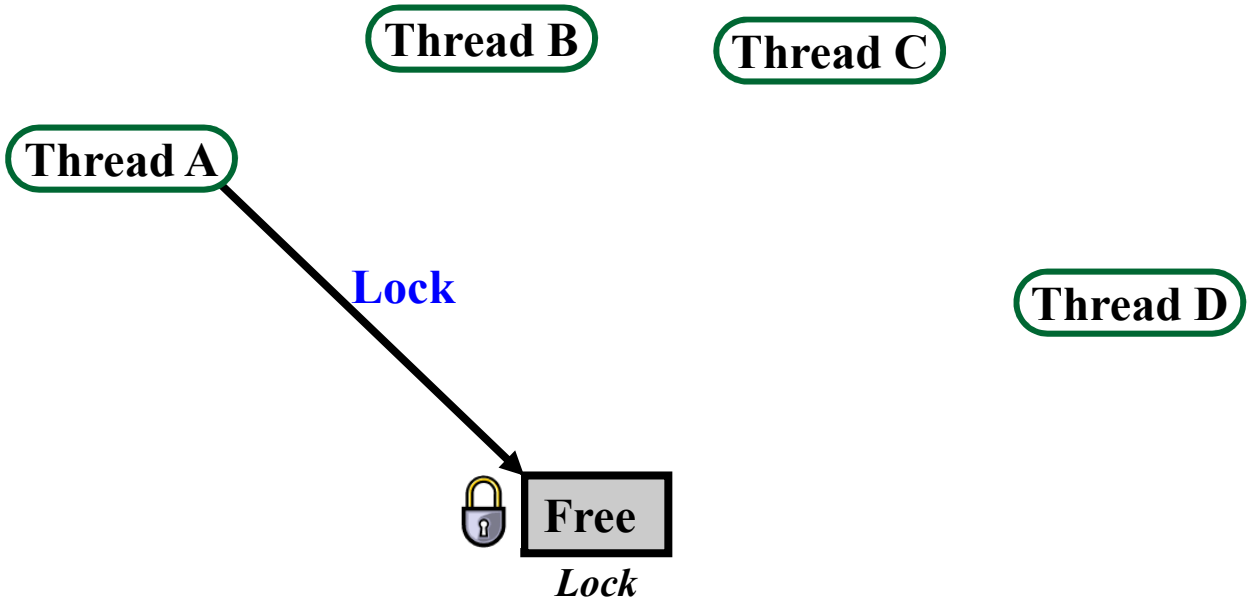
(Donnée, code non réentrant)

- Le temps passé dans une section critique doit être court

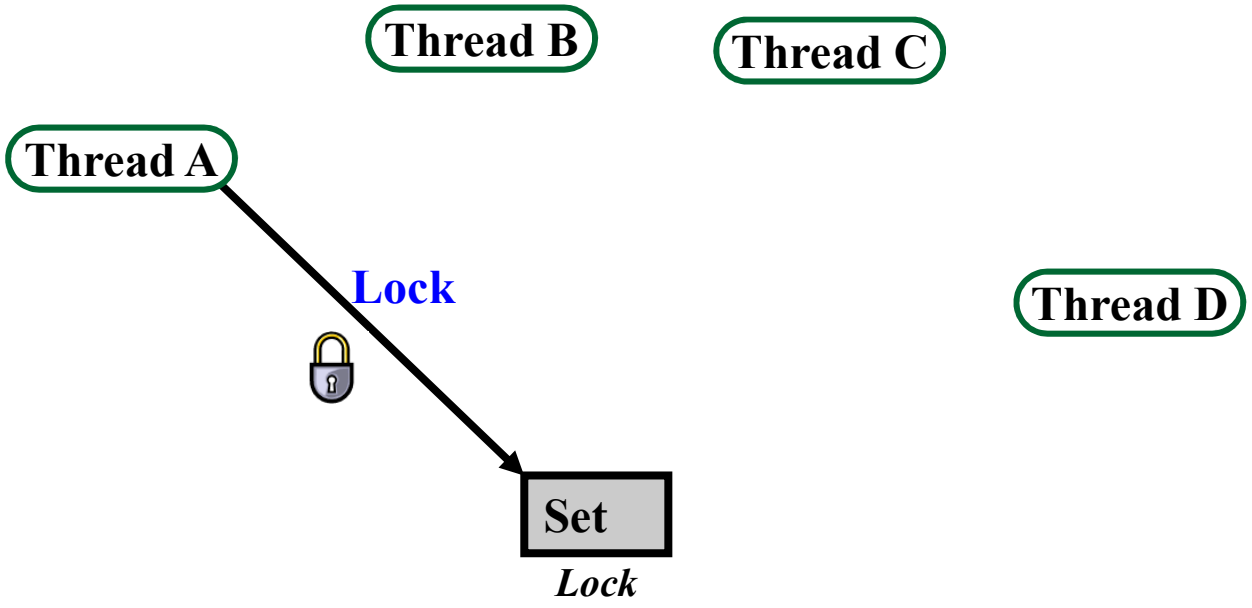
Acquisition et Libération de Verrous (Locks)



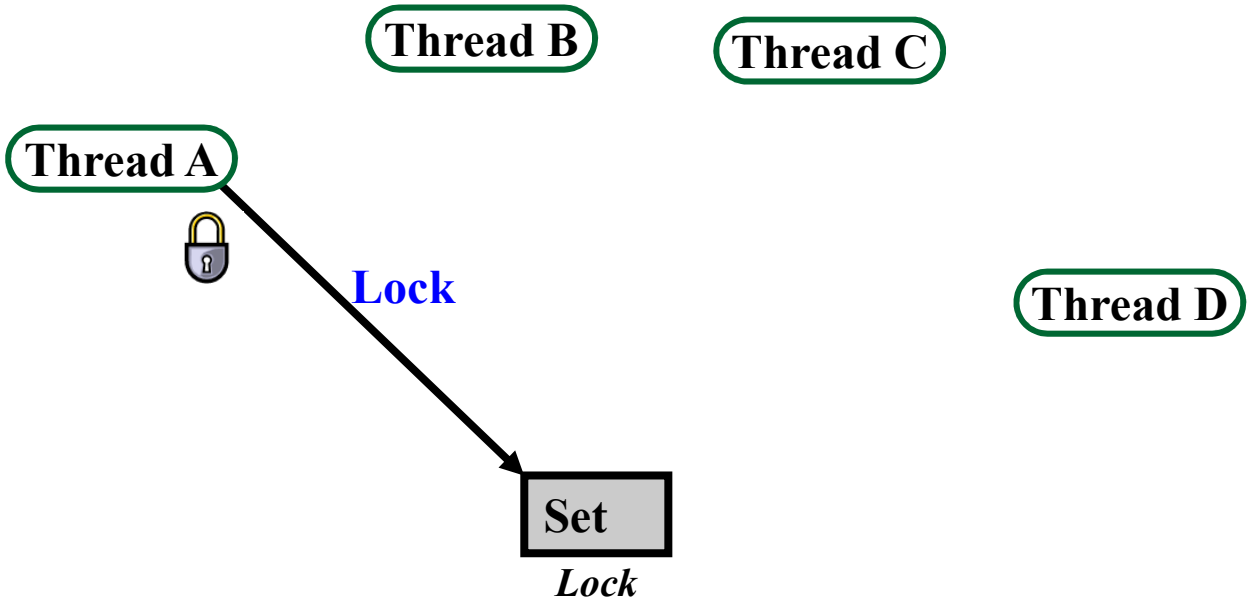
Acquisition et Libération de Verrous (Locks)



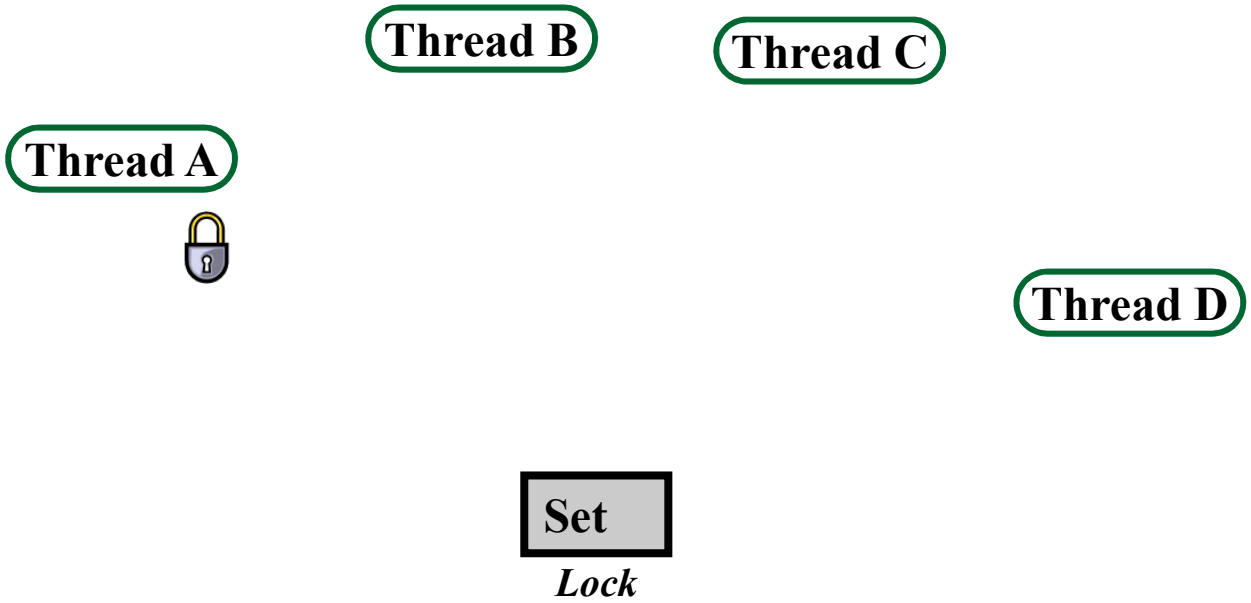
Acquisition et Libération de Verrous (Locks)



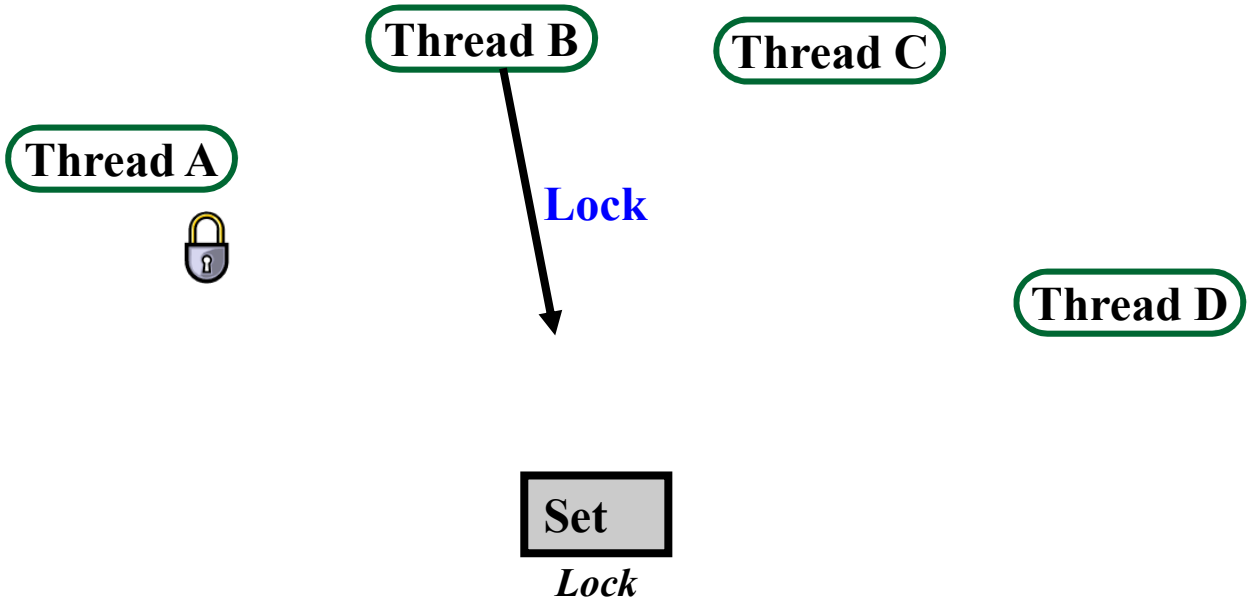
Acquisition et Libération de Verrous (Locks)



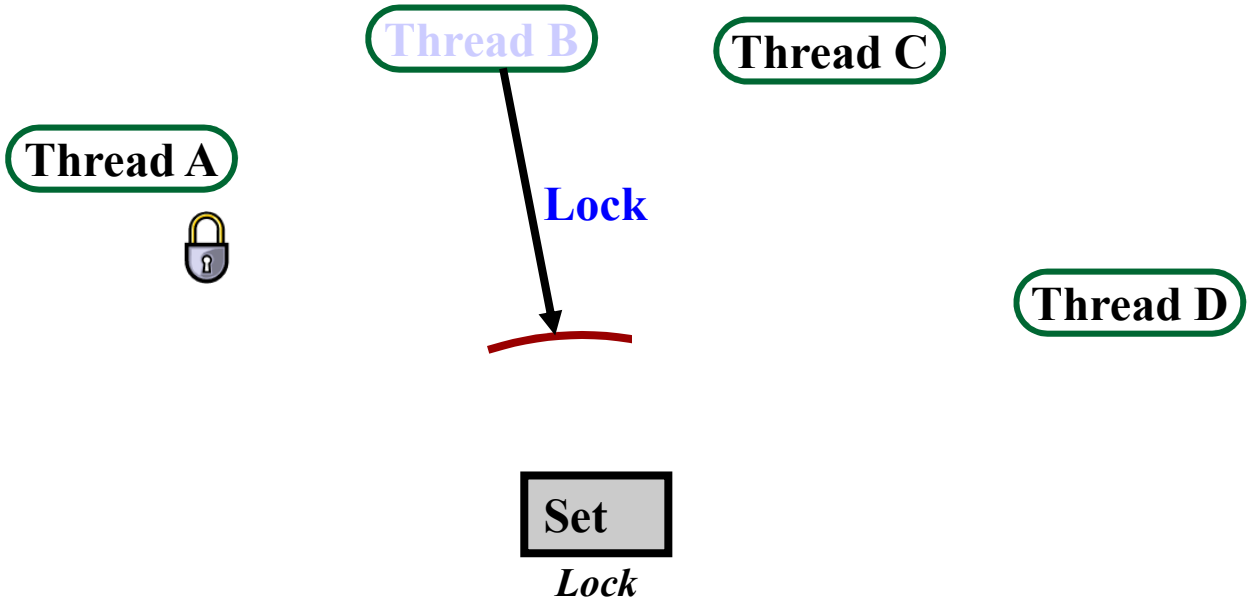
Acquisition et Libération de Verrous (Locks)



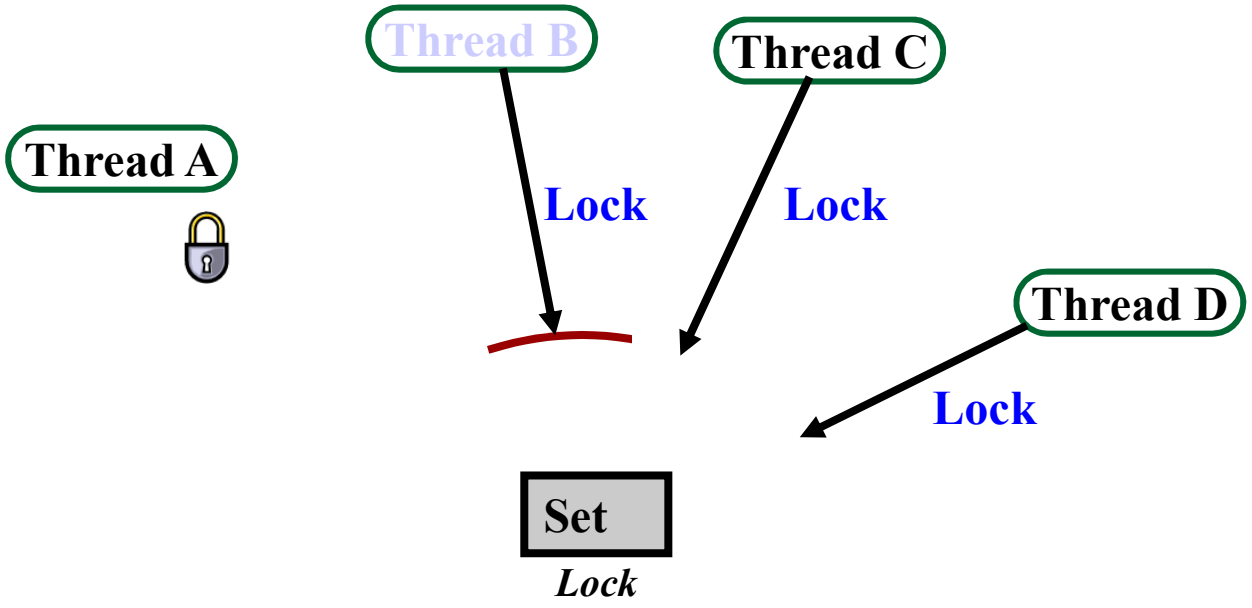
Acquisition et Libération de Verrous (Locks)



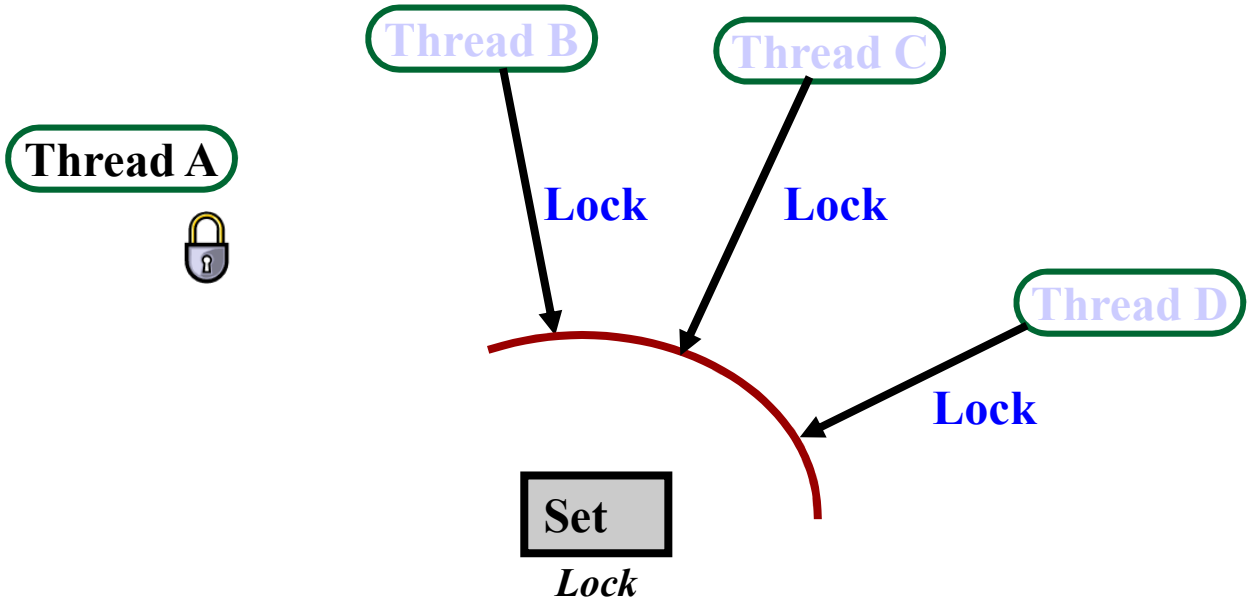
Acquisition et Libération de Verrous (Locks)



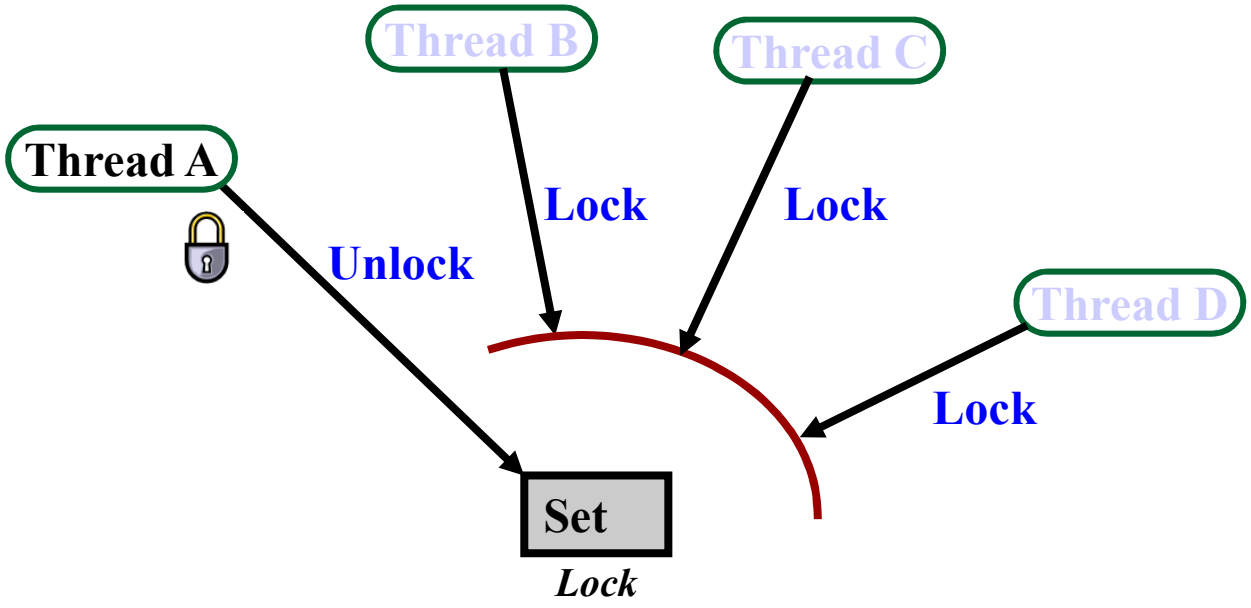
Acquisition et Libération de Verrous (Locks)



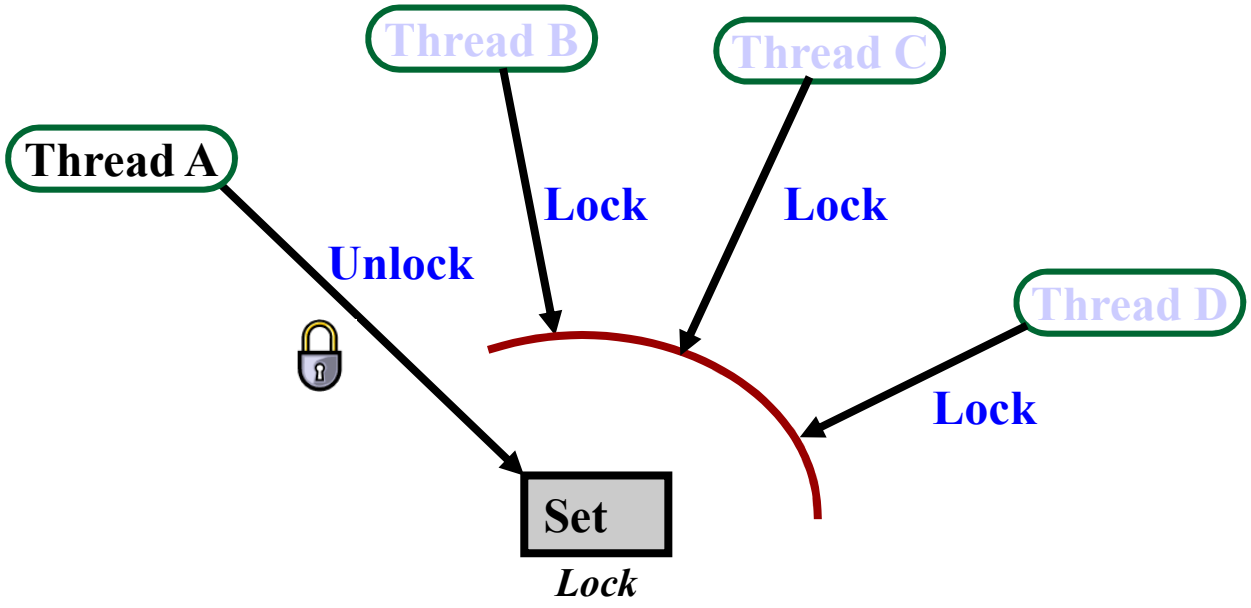
Acquisition et Libération de Verrous (Locks)



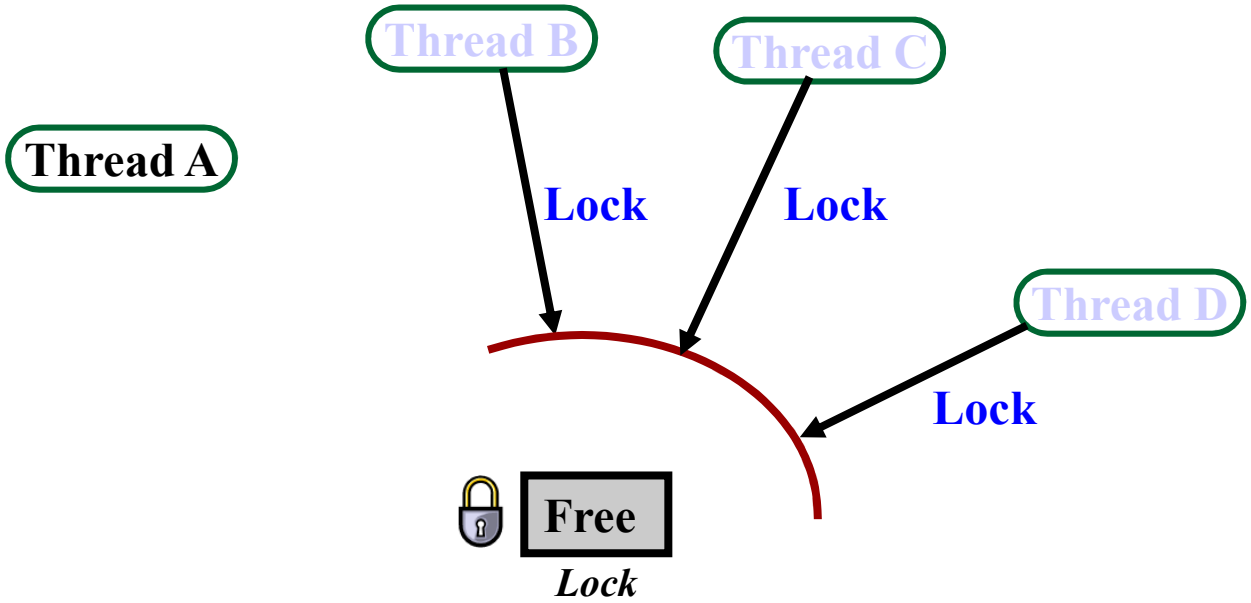
Acquisition et Libération de Verrous (Locks)



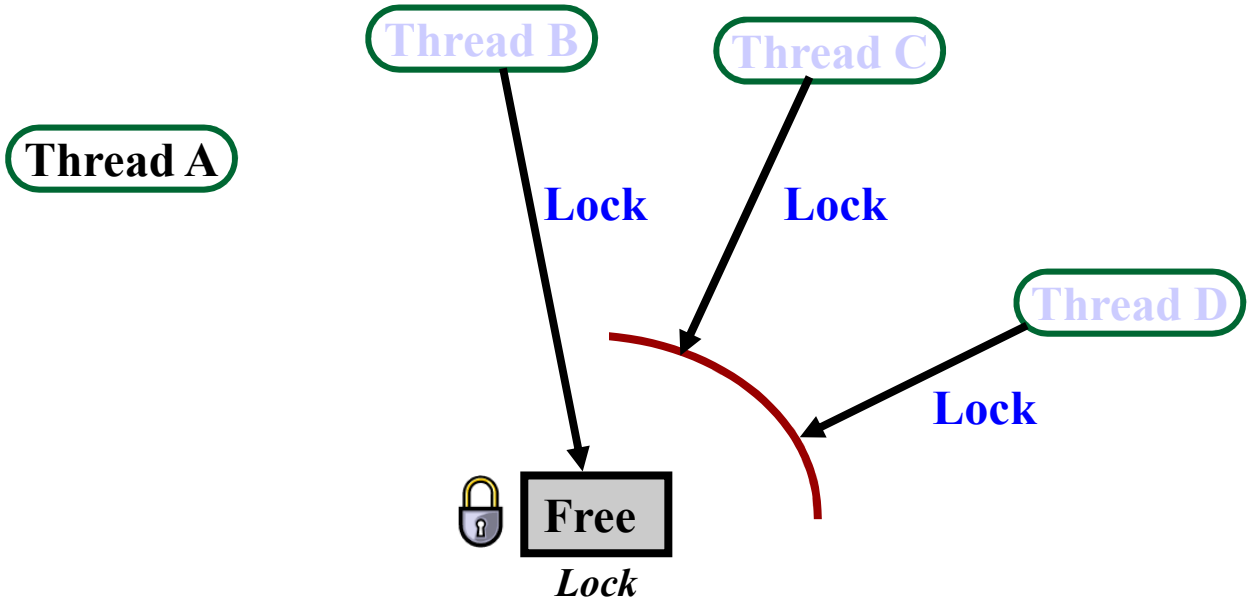
Acquisition et Libération de Verrous (Locks)



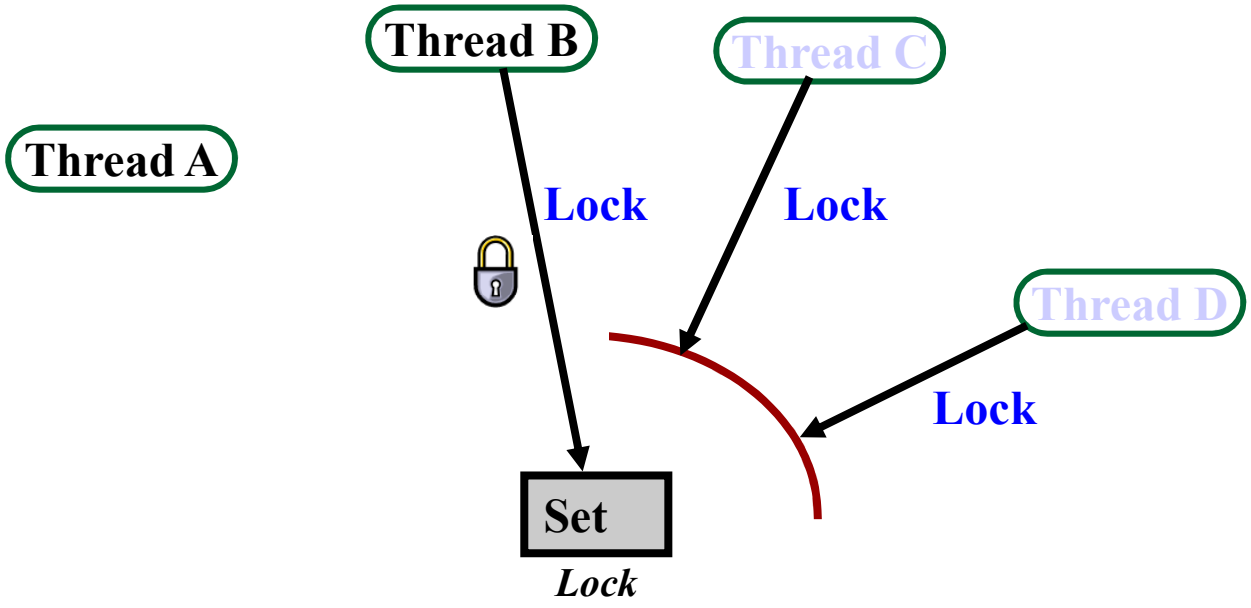
Acquisition et Libération de Verrous (Locks)



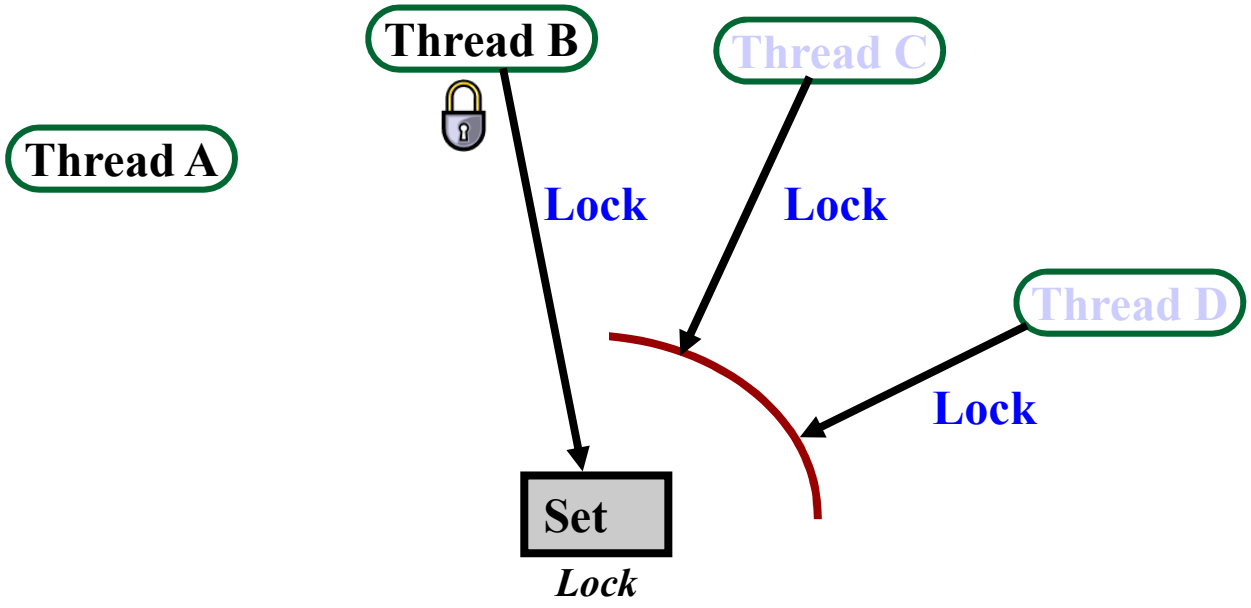
Acquisition et Libération de Verrous (Locks)



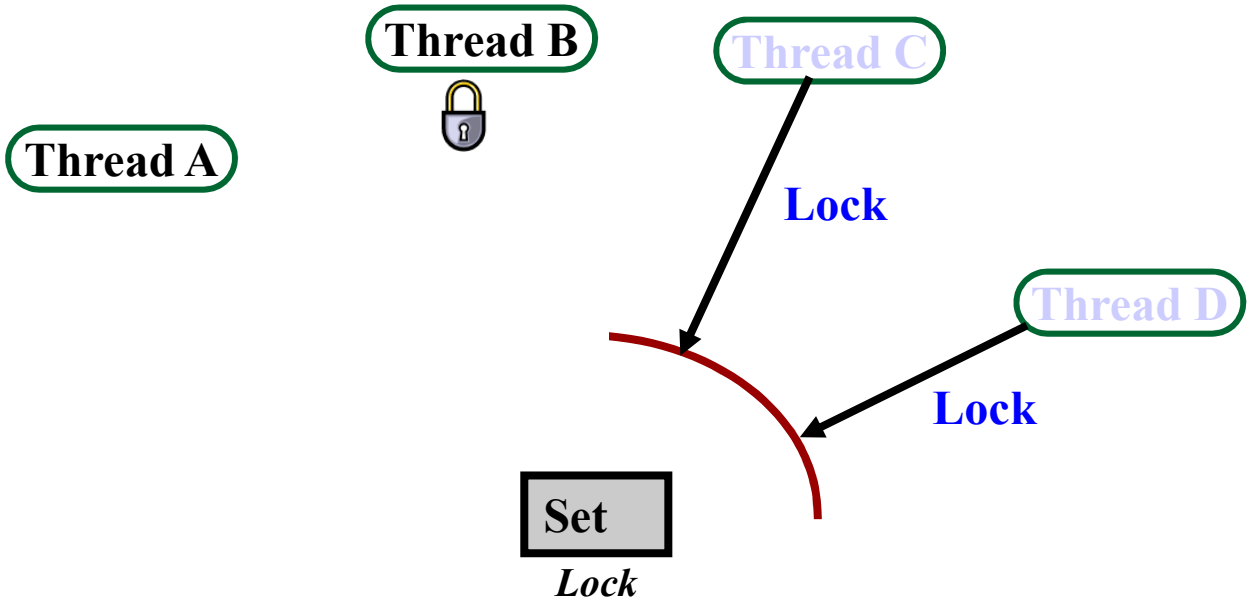
Acquisition et Libération de Verrous (Locks)



Acquisition et Libération de Verrous (Locks)



Acquisition et Libération de Verrous (Locks)



Les Verrous d'exclusion Mutuelle --Mutex

- **Déclaration et initialisation**
  - ***Création statique:*** **pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;**
  - ***Création dynamique:***
    - *pthread_mutex_t mutex;*
    - *pthread_mutex_init(&mutex, & attributs);*
      - Pour les attributs, NULL correspond aux valeurs par défaut
      - les attributs permettent le partage du mutex entre plusieurs processus (via la mémoire partagée).
- **Utilisation d'un verrou :**
  - ***Demande de verrou :***
    - Demande suspensive : **pthread_mutex_lock(&mutex);**
    - Tentative sans blocage: *pthread_mutex_trylock(&mutex); renvoie 0/EBUSY/EINVAL-EFAULT*
  - ***Libération du verrou par le thread qui la pris*** (verrouillé)
    - **pthread_mutex_unlock(&mutex)**; la libération des threads se fait de façon aléatoire.
- **Destruction d'un verrou :**
  - *pthread_mutex_destroy(&mutex);* détruit le mutex qui doit être déverrouillé

Les Mutex

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Les Verrous d'Exclusion Mutuelle -- Exemples

Reprenons l'exemple précédent. Pour en assurer le bon fonctionnement, il faudrait modifier la boucle ainsi :

- **Avant modification :** *for (i=0; i <1000000; i++) val = val + 1;*
- **Après modification :**

```
for (i=0; i <1000000; i++) {
    pthread_mutex_lock (&verrou);
            val = val + 1;
    pthread_mutex_unlock (&verrou);
}
```

Exemple 2 : Deux threads T1,et T2 doivent assurer une contrainte de cohérence sur les données x et y , qui est ici : **x = y**.

```
thread_mutex_t verrou;
int x, y;
```

Thread T1

```
•••
pthread_mutex_lock(&verrou);
        x = x + 1;
        y = y + 1;
pthread_mutex_unlock(&verrou);
•••
```

Thread T2

```
•••
pthread_mutex_lock (&verrou);
        x = 2 * x;
        y = 2 * y;
pthread_mutex_unlock(&verrou)
•••
```

Les Verrous -- Interblocage

- *mutex M1, M2;*
- ***Thread A :***

  Lock(M1); //1. A verrouille M1

  Lock(M2); // 3. A se bloque sur M2

  Thread B :

  Lock(M2); // 2. B verrouille M2

  Lock(M1); / 4. B se bloque sur M1

  - **Solution:**
    - Concevoir sans deadlock -- **imposer un ordre** (acyclique) sur les verrous, qui doivent nécessairement être demandés en respectant cet ordre.
- ***Inversion de priorité :***
  - Soient trois threads A, B, et C, tels que A est de forte priorité, B de priorité moyenne, et C de faible priorité ainsi qu'un verrou m.
    - *A et B sont bloqués*
    - *C s'exécute et verrouille m*
    - *B se réveille et préempte B*
    - *A tente de verrouiller m et se bloque*
    - *B reprend la main*
  - Tant que B ne se bloque pas, C ne peut pas s'exécuter et libérer le verrou nécessaire à la poursuite de A.
  - **Solution** (rarement implantée) : modifier la priorité de C pour le rendre aussi prioritaire que A jusqu'à ce qu'il ait libéré le verrou

Avoiding Deadlocks

- **Establish a hierarchy :** Always lock Mutex_1 before Mutex_2, etc..,.
- Use the trylock primitives if you must violate the hierarchy.

```
{   while (1)
  {   pthread_mutex_lock (&m2);
      if(EBUSY != pthread mutex_trylock (&m1))
        break;
      else
      {    pthread _mutex_unlock (&m1);
           wait_around_or_do_something_else();
      }
  }
  do_real work();     /* Got `em both! */ }
```

Use **lockllint** or some similar static analysis program to scan your code for hierarchy violations.

Synchronisation -- Les Sémaphores

- ✓ les sémaphores peuvent être utilisés par des threads appartenant à différents processus; inclure l'interface (***#include<semaphore.h>***) et le type est ***sem_t***.
- ✓ la libération des threads bloqués se fait de façon aléatoire (!).

Nom de la fonction	Rôle de la fonction
*sem_init (sem_t *S, shared p, unsigned int valeur)*	Initialisation du sémaphore
*sem_destroy (sem_t *S)*	Suppression du sémaphore
sem_wait (sem_t *S)	Attend que la valeur du sémaphore soit positive, puis la décrémente
*sem_trywait (sem_t *S)*	Décrémente le compteur s'il est positif, sinon erreur
sem_post (sem_t *S)	Incrémente le compteur et libère éventuellement un thread.

Synchronisation -- les Variables Conditionnelles

- les variables conditionnelles s'utilisent conjointement à un verrou afin d'éviter des interblocages.
- **Déclaration et initialisation**
  - ***Création statique:*** *pthread_cond_t cond = PTHREAD_COND_INITIALIZER;*
  - ***Création dynamique:*** *pthread_cond_init(&cond, &attributs);*
    - Pour les attributs, NULL = valeur par défaut
    - Les attributs permettent le partage des variables de condition entre plusieurs processus (lourds - via la mémoir partagée).
- **Utilisation**
  - ***Attente de la condition :*** *pthread_cond_wait(&cond, &mutex);*
    - L'attente est toujours associée à un mutex
    - Le verrou mutex est libéré au moment de la mise en attente (blocage)
  - ***Signalisation de la condition***: **pthread_cond_signal(&cond),** ou encor **pthread_cond_broadcast(&cond);**
    - Une (tous) thread(s) en attente sur la condition sont réveillés
    - Ils sont alors à nouveau en compétition pour le mutex (pour le réacquérir).
- **Destruction d'une variable condition :** *pthread_cond_destroy(&cond);*

Synchronisation -- les Variables Conditionnelles (suite)

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Les Variables Conditionnelles -- Exemple (Producteur/Consommateur)

```
#include <pthread.h>
#include <string.h>
#include ''prodcons.h''
static char *buffer;   /*Tampon partagee de taille =1*/
static pthread_cond_t libre, plein;                static pthread_mutex_t protect;
void init_prodco (void)   /* ----- Initialise le producteur/consommateur  ----------*/
{    pthread_mutex²_init(&protect, NULL); pthread_cond_init(&libre, NULL);
     pthread_cond_init(&plein,  NULL);     buffer = NULL; }
```

```
void *deposer(char *msg)
{ pthread_mutex_lock(&protect);
  while ( buffer != NULL)
     pthread_cond_wait(&libre, &protect);
  buffer = strdup(msg);
 pthread_cond_signal(&plein);
 pthread_mutex_unlock(&protect);
 return NULL;
}
```

```
char *retirer(void)
{ char *result;
pthread_mutex_lock(&protect);
 while (buffer == NULL)
            pthread_cond_wait(&plein, &protect);
 result = buffer; buffer = NULL;
pthread_cond_signal(&libre);
pthread_mutex_unlock(&protect);
return result; }
```

Les barrières avec Mutex et Variables conditionnelles

- *A barrier holds a thread until all threads participating in the* barrier have reached it.
  - Barriers can be implemented using a counter, a mutex and a condition variable.
  - A single integer is used to keep track of the number of threads that have reached the barrier.
  - If the count is less than the total number of threads, the threads execute a condition wait.
- The last thread entering the barrier (and setting the count to be the same as the number of threads) wakes up all the threads using a condition broadcast.

Les barrières Pthread API

```
typedef struct {
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed;
    int count;
} barrier_t;


void barrier_init(barrier_t *b) {
    pthread_mutex_init(&(b->count_lock), NULL);
    pthread_cond_init(&(b->ok_to_proceed), NULL);
    b->count = 0;   }
```

Les barrières Pthread API

```
void barrier (barrier_t *b,  int num_threads)
{
    pthread_mutex_lock(&(b->count_lock));
    b->count++;
    if (b->count == num_threads) {
        b->count = 0;
        pthread_cond_broadcast(&(b->ok_to_proceed));
    }
    else {
        while (b->count != 0)
            pthread_cond_wait(&(b->ok_to_proceed), &(b->count_lock));
        }
    pthread_mutex_unlock(&(b->count_lock));
}
```

Synchronisation - Récapitulatif (1)

- **Verrou d'exclusion mutuelle (mutex-lock)**
  - l'outil le plus rapide et le moins consommateur de mémoire. **Il doit être rendu par le thread qui l'a pris** . Il s'utilise surtout pour sérialiser l'accès à une ressource ou assurer la cohérence des données.

- **Sémaphore**
  - consomme plus de mémoire et s'utilise dans les cas où on se synchronise sur l'état d'une variable, plutôt qu'en attendant une commande venue (cond_signal) d'un autre thread. Il n'exige pas que le verrouillage/déverrouillage soit fait par le même thread.

- **Variables conditionnelles (condition variables)**
  - Une variable conditionnelle s'utilise pour attendre l'occurrence d'un événement.
  - Le verrou qu'on doit lui associer sert à gérer l'exclusion mutuelle sur les variables internes liées à cette variable conditionnelle (compteur).
  - Ce verrou est automatiquement rendu par le système lors de l'appel à cond_wait et repris dès la sortie de cette fonction. Il doit être rendu par l'utilisateur après cette sortie.

Synchronisation - Récapitulatif (2)

- **Visibilité:**
  - les outils de synchronisation peuvent être vus par des threads appartenant à des processus DIFFERENTS :
- **Passage à l'état bloqué, sortie de cet état :**
  - L'appel à cond_wait est **toujours** bloquant (à la différence de lock ou sem_wait)
  - cond_signal ne fait qu'essayer de débloquer un thread, le signal est **perdu** si aucun thread n'est bloqué lors de son émission (alors que V incrémente un compteur)
  - pour éviter les interblocages, utiliser les verrous ou les sémaphores suivant un ordre fixe, ou utiliser les variables conditionnelles
- **Performances :**
  - Attention à l'inversion de priorité

Conclusions sur IPC threads: threads ou non threads?

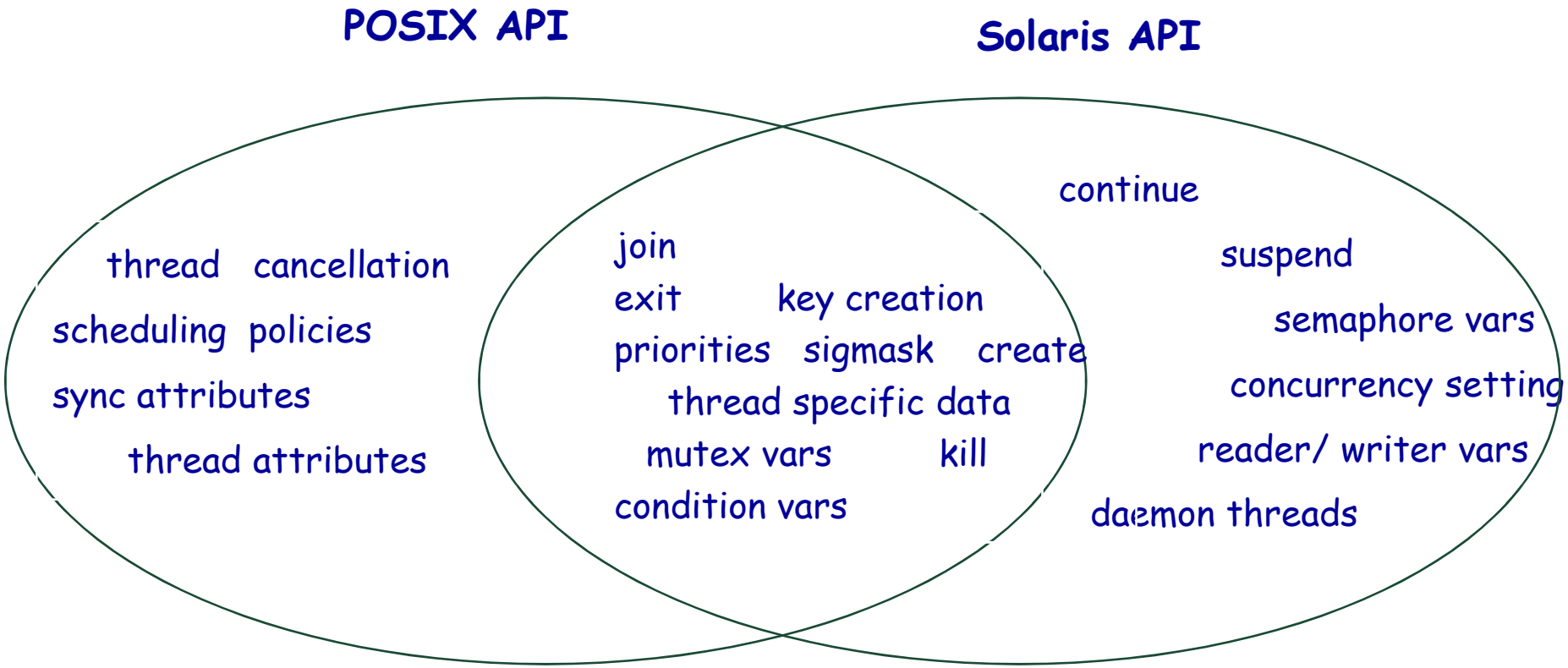
- ✓ Profiter pleinement des multicores
- ✓ Améliorer le "Throughput" → **Implémenter des E/S Asynchrones**
- ✓ Bien profiter des propriétés du noyau système
- ☹ Si toutes les opérations sont CPU Bound → ne pas aller plus loin dans le multithreading
- ☹ La création d'un thread n'est pas coûteuse, mais elle n'est pas gratuite
  - ➢ **1 thread qui a seulement 4-5 lignes de code devrait être très utile!**

Annexe: The APIs

Different Thread Specifications

Functionality	UI Threads	POSIX Thteads	NT Threads	OS/2 Threads
Design Philosophy	Base	Near-Base	Complex	Complex
Primitives	Primitives	Primitives	Primitives	
Scheduling Classes	Local/ Global Local/Global	Global	Global	
Mutexes	Simple	Simple	Complex	Complex
Semaphores	Simple	Simple	Buildable	Buildable
R/W Locks	**Simple**	Buildable	Buildable	Buildable
Condition Variables	Simple	Simple	Buildable	Buildable
Multiple-Object	Buildable	Buildable	Complex	Complex
Synchronization				
Thread Suspension	Yes	Impossible	Yes	Yes
Cancellation Buildable	Yes	Yes	Yes	
Thread-Specific Data	Yes	Yes	Yes	Yes
Signal-Handling				
Primitives	Yes	Yes	n/a	n/a
Compiler Changes				
Required	No	No	Yes	No
Vendor Lib. MT-safe? Most		Most	All?	All?

POSIX and Solaris API Differences



COMMUNICATION INTERPROCESSUS

Plan de la communication interprocessus

1. Définitions et Propriétés de communication
2. Pipes Unix
3. Les signaux
4. Les files de messages (MSQ)

Moyens de communication

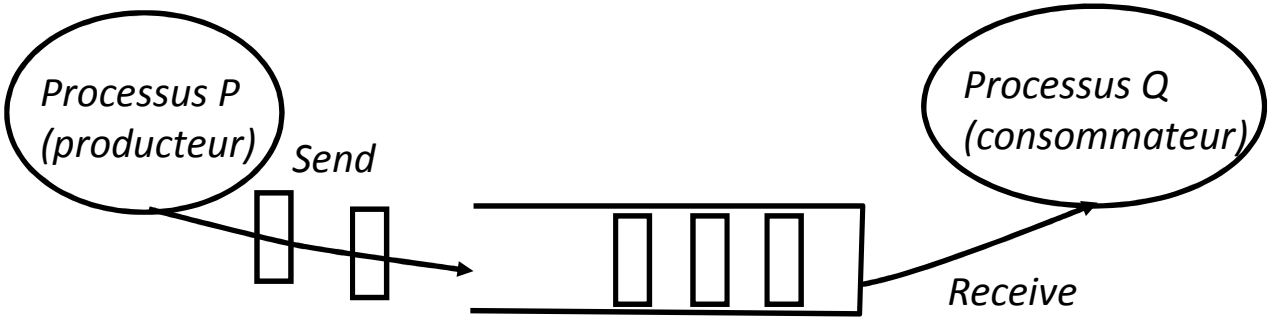
- ✓ Moyens de communication interprocessus intra-machine (UNIX)
  - ➢ Appartenant au système de gestion de fichiers
    - Pipes anonymes
    - Pipes nommés
  - ➢ Famille des IPC (Inter Processus Communication)
    - Files de messages (MSQ)
  - ➢ Les signaux
- ✓ Moyens de communication interprocessus inter-machines
  - ➢ sockets, RPC, RMI, …

Communication par Messages

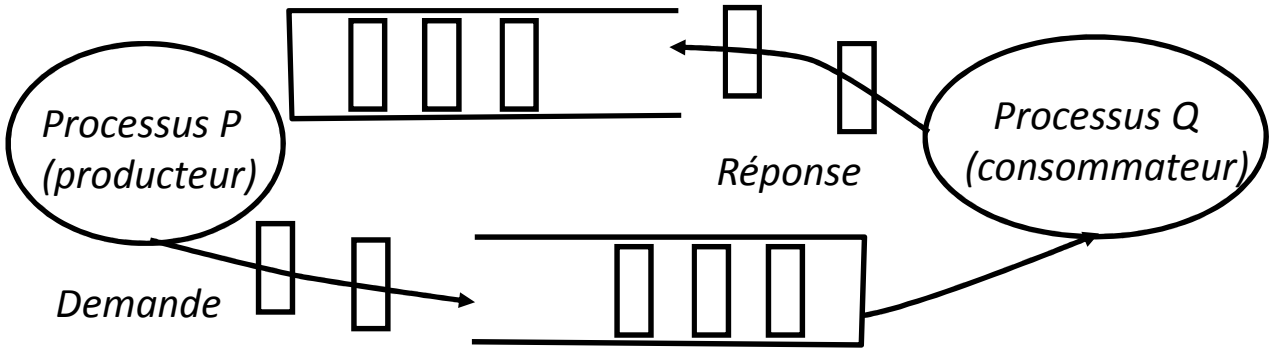
- **Deux opérations fondamentales :**
  - Emettre : **send(destination, message)**
  - Recevoir : receive(source, message)
- Les messages sont de tailles variables ou fixes
- L'envoi et la réception peuvent être soit directes entre les processus à travers l'identifiant du destinataire, soit indirectes par l'intermédiaire d'une boite aux lettres.
- Il existe des variantes de ***send/receive*** en fonction des sémantiques :
  - **send(destination, message)** : bloquante ou non?
  - **receive(source, message)** : synchrone ou non?

Styles de Communication par Messages

- ✓ **Unidirectionnel** : les messages suivent un seul sens (les pipes d'Unix ou Producteur/Consommateur).



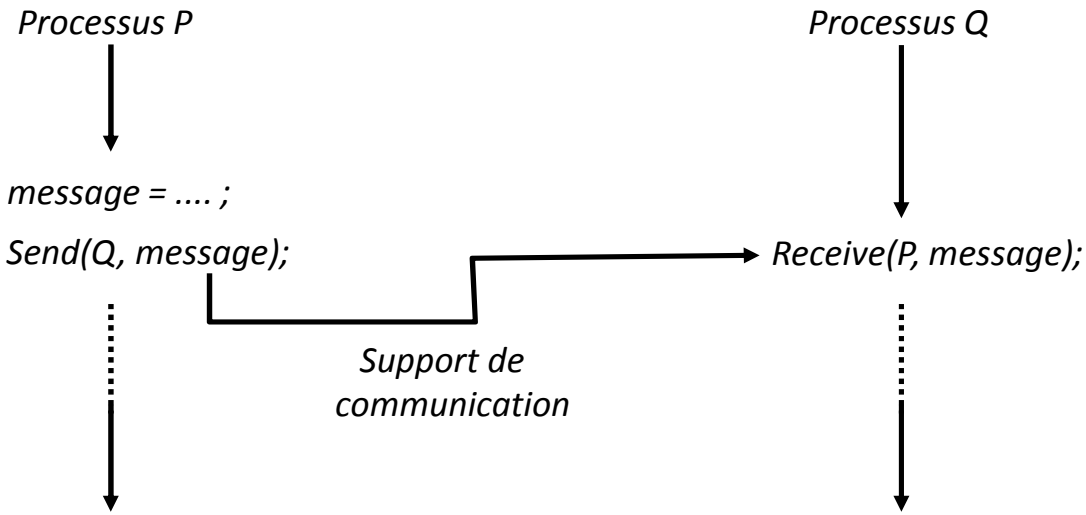
- ✓ **Bidirectionnel** : les messages suivent un cercle (Client/Serveur, Appel de procédure à distance -- RPC)



Propriétés Fondamentales de la Communication
-- Identification 1 --

✓ **Comment sont spécifiées destination et source?**

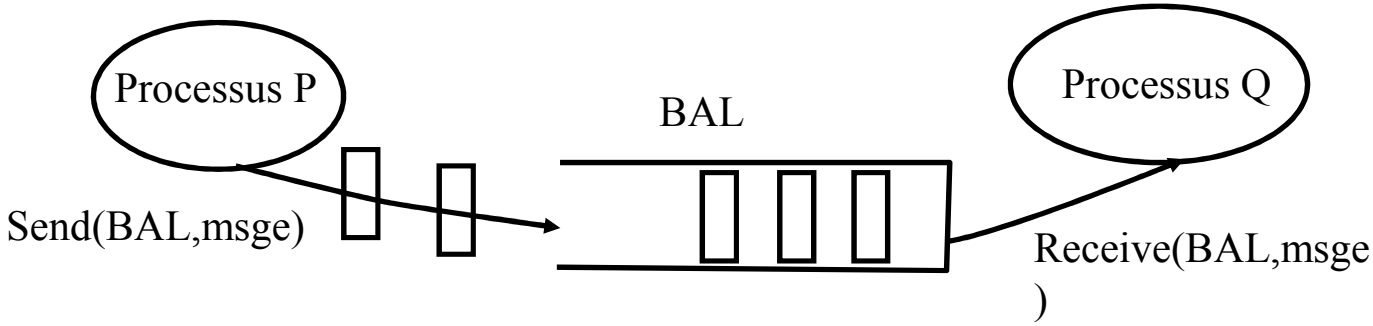
- Chaque processus écrit le nom du destinataire à qui le message est envoyé ou bien le nom de la source à partir de laquelle le message est reçu



- **Nom processus = processus@machine{.domaine}**
- **Multicast/Broadcast** : envoi à tous/un groupe de destinataires
  - **Jockers (Wildcard) :** en PVM **send(-1, msge); receive(-1, -1)**

Propriétés Fondamentales de la Communication
-- Identification 2 --

- **Communication par boite aux lettres (file de messages)**



✓ **Principe de la boite aux lettres :**

- BAL pleine → l'émetteur se bloque jusqu'à ce qu'il y aurait de la place
- BAL vide → le récepteur se bloque jusqu'à ce qu'un message soit placé

✓ Prévoir une BAL par classe de processus

⇨éviter la réception d'un message à partir d'un processus destinataire qcq.

Propriétés Fondamentales de la Communication -- Mode de communication

- **Envoi asynchrone:** en général **send** non bloquante:
  - ⇨ Le processus émetteur continue son exécution même si le récepteur n'a pas retiré son message.
  - ⇨ Envoi de plusieurs messages : ordre de retrait = ordre d'émission
- **Réception synchrone: receive** bloquante
  - ⇨ Attente jusqu'à ce qu'un processus source envoit un message
- **Rendez-Vous : send** et **receive** sont bloquantes :
  - ⇨ Le premier qui arrive attend l'autre; quand l'émetteur et le récepteur correspondant sont en attente, le message est transféré et les deux sont permis de continuer.

Propriétés Fondamentales de la Communication
-- Tampons/Taille des messages --

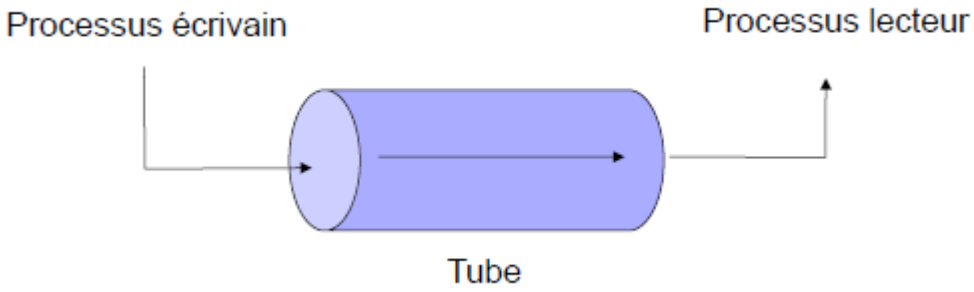
- **Manipulation des tampons:**
  - ⇨ Les messages sont copiés directement de la mémoire de l'émetteur?
  - ⇨Est-ce qu'ils sont copiés d'abord dans un certain tampon système entre les deux processus?
- **Taille des messages :**
  - ⇨ Taille limite de message à transférer? court? long? de taille fixe?
- **Conclusion sur les propriétés de la communication :**
  - ⇨ Toutes les propriétés ne sont pas indépendantes:
    - ↳ Par exemple la primitive ***send*** non bloquante n'est généralement accesssible qu'à travers la manipulation des tampons.

Problème du Producteur/Consommateur (Tampon borné)
-- Synchronisation par messages --

Processus Producteur	***Processus Consommateur***
{ *int x;* *message msg;* *While true {* *produire_objet(& x);* *receive(consommateur, &msg);* *packet_msge(&msg, x);* *send(consommateur, &msg);* *}* *}*	*{* *int x, i;* *message msg;* *for (i=0; i<N; i++)* *send(producteur, &msg);* *While true {* *receive(producteur, &msg);* *extraire_objet(&msg, &x);* *consommer_objet(x);* *send(producteur, &msg);* *}* *}*

Communication par PIPES (Unix-Linux)

- ✓ **Unix est un système basé sur le passage de messages :**
  - ⇨ Un shell Unix: le signe "|"
  - ⇨ API: appel système "pipe"
  - ⇨ Un pipe permet à 2 processus de s'exécuter en même temps; le 1er fournissant les données que le snd exploite au fur et à mesure de leur production.



Les PIPES: Definition et types

- ✓ Un pipe est un canal unidirectionnel en mode flots d'octets
  - ⇨ Fichier spécial à usage interne pour la communication
  - ⇨ Un processus, producteur, fournit (écrit) des données dans le pipe, l'autre processus, consommateur, l'exploite (lit).
  - ⇨ L'information disparait (implicement/explicitement) après lecture.
- ✓ Deux types de pipes (SGF):
  - ⇨ Les pipes anonymes: entre processus avec lien de parenté.
  - ⇨ Les pipes nommés (FIFO): entre processus sans lien de parenté.
- ✓ Unix offre 2 primitives d'émission et de réception :
  - ➢ **Write(int desc, char *buf, int taille)**: non bloquante
  - ➢ **Read(int desc, char *buf, int taille):** bloquante

Les PIPES anonymes-- Exemple

```
#include<errno.h>
#include<stdio.h>
#include<unistd.h>  /*contient l'appel système pipe */
void main (void) {
    int tube[2]   /* deux descripteurs de pipe : 0 --> lecture et 1 --> écriture */
    char car1;  /* caractère de l'alphabet*/
    int ret;  /*Valeur de retour du fils*/

    pipe(tube);   /* Création du tube -- tampon interne */
    switch (fork())  /*Création d'un processus fils*/
        case -1 : { perror("impossible de créer un processus);  exit(2); }
        case 0 : { close(tube[1]); /* le fils ne peut écrire dans le tube*/
                    while (read(tube[0], &car1,1)>0)  /* le fils lit le tube*/
                        printf("%c\n", car1);
                   close(tube[0]) ; /* fermer le tube en lecture*/ exit (0);}
        default : {close(tube[0]) /* Le père ne lit pas */
                    for (car1='a'; car1<='z'; car1++)
                        write(tube[1]), &car1, 1); /* Le père écrit des lettres dans le tube */
                   close(tube[1]); /*fermer le tube en écriture */
                   wait(&ret);  /* Blocage en attente de la fin du fils */
                   exit(0); } }
```

Pipes Unix Nommés --FIFO

- Caractéristiques communes aux tubes anonymes:
  - ⇨ communication entre processus s'exécutant sur une même machine
  - ⇨ fichier particulier
  - ⇨ file de messages en mémoire
  - ⇨ pointeurs gérés automatiquement

Pipes Unix Nommés: caractéristiques

- Différences avec les tubes anonymes:
  - fichier portant un nom
  - filiation non nécessaire
  - création par la fonction SGF **mknod()/mkfifo()**
  - ouverture par open()
  - un seul descripteur par ouverture de tube
  - fichier persistant

Pipes Unix Nommés

- Utilisation d'un pipe nommé (**FIFO**)
  - Créer le pipe nommé
  - Ouvrir le pipe en lecture et en écriture
  - Lire et écrire dedans
  - Fermer les descripteurs
  - supprimer le pipe (*rm fichier* sur le shell).
- Pour pouvoir lire ou écrire dedans, il faut que le tube nommé soit ouvert à la fois en lecture et en écriture.
- Si ce n'est pas le cas les opérations de lecture/écriture sont bloquantes.

Ouverture d'un fichier

desc= open(nom_du_fichier, mode)

- ouverture en lecture si mode = O_RDONLY
- ouverture en écriture si mode = O_WRONLY
- ouverture en maj si mode = O_RDWR
- ouverture bloquante / non bloquante mode =O_NDELAY

mode	O_RDONLY	O_WRONLY
O_NDELAY	ouverture sans attente	ouverture avec retour code erreur si aucune ouverture en lecture
sinon	processus bloqué jusqu'à ouverture en écriture par un autre processus	processus bloqué jusqu'à ouverture en lecture par un autre processus

mkfifo()

```
#include <sys/types.h>

#include <sys/stat.h>
```

```
int mkfifo ( const char *fileName, mode_t mode);
int mknod (nom_du_fichier, accès+S_IFIFO)
```

- L'appel système mkfifo permet de créer un tube nommé (ou fifo). Cela a pour effet de créer un fifo dans le système de fichiers.
- Une fois créé le fifo peut être ouvert en lecture ou en écriture.
- ⇨ Les lectures ou écritures seront bloquantes tant que l'ouverture en lecture et en écriture n'aura pas eu lieu.
- Valeur renvoyée: retourne 0 en cas de succès ou la valeur -1 en cas d'erreur..

Exemple de tube nommé

```
/* Processus ecrivain */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
        main()
        {       mode_t mode;
                int tub;

                mode = S_IRUST | S_IWUSR;
                mkfifo ("fictub",mode) /* création fichier FIFO */
                tub = open("fictub",O_WRONLY) /* ouverture fichier */
                write (tub,"0123456789",10); /* écriture dans fichier */
                close (tub);
                exit(0);}
```

Exemple (suite)

```
/* Processus lecteur */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
        main()
                {int tub;
                char buf[11];
                tub = open("fictub",O_RDONLY) /* ouverture fichier */
                read (tub,buf,10); /* lecture du fichier */
                buf[11]=0;
                printf("J'ai lu %s\n, buf);
                close (tub);
                exit(0); }
```

Exercice pipe anonyme

Enoncé: Ecrire un programme, qui lit des caractères sur l'entrée standard et envoie dans un pipe les lettres et chiffres à son processus fils. Le processus fils compte les lettres et les chiffres et affiche les résultats à la fin.

N. B. Le processus père attend la terminaison du fils pour s'arrêter.

Correction Exercice pipe anonyme

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <ctype.h>
 int main (void) {
        pid_t retour ;
        int tube[2], lettre = 0, chiffre = 0 ;
        char k ;
        pipe (tube) ;
         switch (retour = fork ()) {
        case -1 : perror ("Creation impossible") ;
                exit(1) ;
        case 0 : printf ("processus fils\n") ;
        /* le tube est ici ferme en ecriture: le dernier descripteur ouvert */
        /* en ecriture sur le tube sera dans le processus pere. Quand celui */
```

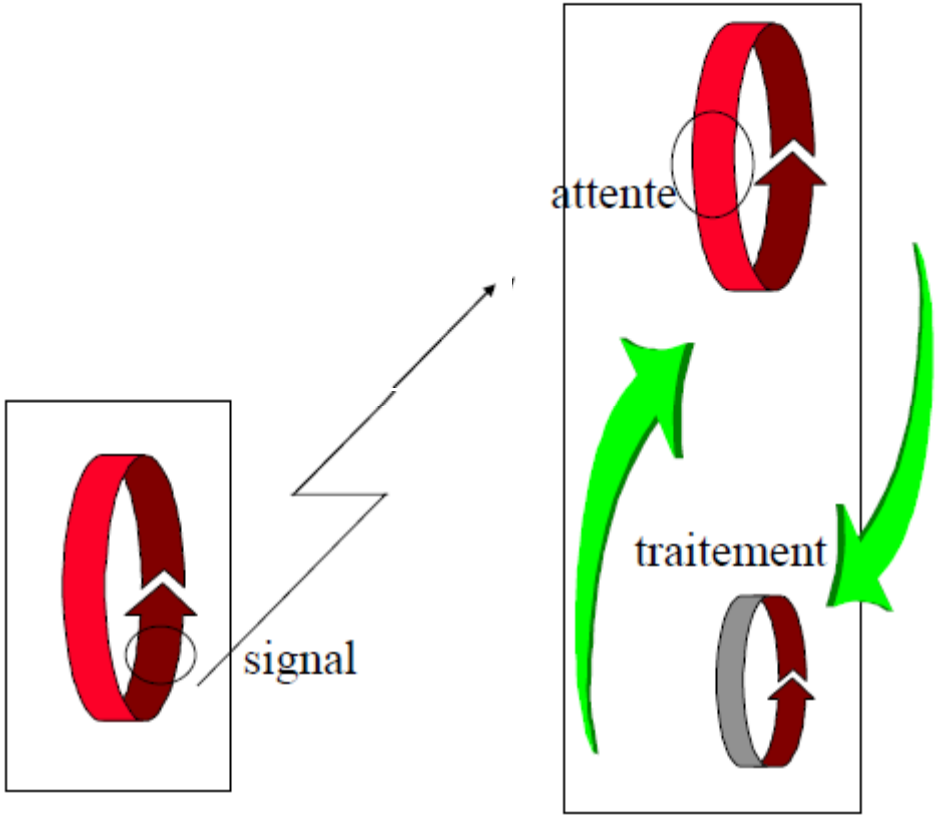
Correction Exercice pipe anonyme (suite)

```
        /* ci fermera ce descripteur, le read effectué par le fils renverra 0 */
        close (tube[1]) ;
        while (read (tube[0], &k, 1) >0)
                if (isdigit (k)) chiffre++ ; else lettre++ ;
        printf ("%d chiffres recus\n", chiffre) ;
        printf ("%d lettres recues\n", lettre) ;
        exit (0) ;
default : printf ("pere: a cree processus %d\n", retour) ;
        close (tube[0]) ;
        while (read (0, &k, 1) >0)
                if (isalnum(k)) write (tube[1], &k, 1) ;
        /* le tube est ici ferme en ecriture : un read sur le */
        /* tube vide retournera 0 dans le processus fils */
        close (tube[1]) ;
        wait (0) ;
        printf ("pere: a recu terminaison fils\n") ; } }
```

Les signaux

Définition

- Qu'est-ce qu'un signal?
  - interruption d'un processus
- Fonctions utiles
  - envoi du signal
  - attente du signal
  - traitement à effectuer



Définition (suite)

- **Signal = interruption logicielle**
  - ⇨ événement asynchrone
  - ⇨ destiné à un processus
  - ⇨ émis par un autre processus ou par le système
- Les interruptions logicielles ou signaux sont utilisées pour communiquer avec un processus.

Gestion des Signaux

- Réaction à un événement sans être obligé d'en tester en permanence l'arrivée.
- Un signal est délivré à un processus lorsque celui-ci le prend en compte au cours de sa propre exécution.
- Les processus peuvent indiquer au système ce qui doit se passer à la réception d'un signal.
- Les signaux ne servent pas à échanger des données

Gestion des Signaux (suite)

- Il est possible :

  1. d'ignorer le signal;
  2. de le prendre en compte par l'exécution d'une fonction particulière — qualifiée de **handler** — créée pour l'occasion ;
  3. de laisser le système appliquer le comportement par défaut (qui —en général — consiste à tuer le processus).

- **Certains signaux ne peuvent être ni ignorés, ni capturés.**

Mécanisme de traitant de signaux

- Réaction à la réception d'un signal
- Traitant de signal (handler)
- Installation du traitant
- Fonction appelée de manière asynchrone à la réception d'un signal
- **Reprise de l'exécution à son point d'interruption**

Communication bas niveau

- Transporte peu d'information: numéro de signal
- Utilisée pour informer d'un événement
- Accord entre émetteur et récepteur sur la sémantique de l'événement

Liste des signaux Linux

Listes des signaux : `man -k signal` ou `man 7 signal`

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGABRT	7) SIGBUS	8) SIGFPE
9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	16) SIGSTKFLT
17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU
25) SIGXFSZ	26) SIGVTALRM	27) SIGPROF	28) SIGWINCH
29) SIGIO	30) SIGPWR	31) SIGSYS	34) SIGRTMIN
35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3	38) SIGRTMIN+4
39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12
47) SIGRTMIN+13	48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14
51) SIGRTMAX-13	52) SIGRTMAX-12	53) SIGRTMAX-11	54) SIGRTMAX-10
55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7	58) SIGRTMAX-6
59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX		

Liste des signaux Linux (suite)

Signal	Valeur	Action	Commentaire
SIGHUP	1	Term	Déconnexion détectée sur le terminal de contrôle ou mort du processus de contrôle.
SIGINT	2	Term	Interruption depuis le clavier.
SIGQUIT	3	Core	Demande « Quitter » depuis le clavier.
SIGILL	4	Core	Instruction illégale.
SIGABRT	6	Core	Signal d'arrêt depuis abort(3).
SIGFPE	8	Core	Erreur mathématique virgule flottante.
SIGKILL	9	Term	Signal « KILL ».
SIGSEGV	11	Core	Référence mémoire invalide.
SIGPIPE	13	Term	Écriture dans un tube sans lecteur.
SIGALRM	14	Term	Temporisation alarm(2) écoulée.
SIGTERM	15	Term	Signal de fin.

- Les effets de la réception de ces signaux sur un processus sont :
  - **Term**: le processus se termine ;
  - **Core**: Term + copie de la mémoire dans un fichier core ;
  - **Stop**: suspend l'exécution du processus.

Utilisation des signaux

- Un processus peut **détourner** les signaux reçus et modifier son comportement par l'appel de la fonction:

```
void *signal(int signal, void (*fonction)(int))
```

  - ⇨ Le dernier argument est un pointeur sur une **fonction** qui implémente le nouveau comportement.

Les signaux -Exemple 1

```
   #define _XOPEN_SOURCE
   #include <stdio.h>
   #include <stdlib.h>
   #include <unistd.h>
   #include <signal.h>

   void
gestionnaire (int numero_signal)
{
   fprintf (stdout, "\n %u a reçu le signal %d\n", getpid( ).
   numero_signal);
}

   int
main (void) {
   int i;

   for (i = 1; i < _NSIG; i++)
      if (signal (i, gestionnaire) = SIG_ERR)
         fprintf (stderr, "%u ne peut capturer le signal %d\n",
                          getpid( ), i);
   while (1) {
      pause( );
   }
}
```

Les signaux –Exemple1- Exécution

```
$ ./exemple_signal_2
6745 ne peut capturer le signal 9
6745 ne peut capturer le signal 19
(Contrôle-Z)
6745 a reçu le signal 20
(Contrôle-Z)
[1]+ Stopped ./exemple_signal_2
$ ps 6745
PID TTY STAT TIME COMMAND
6745 p5 T 0:00 /exemple_signal_2
$ fg
./exemple signal 2

6745 a reçu le signal 18
(Contrôle-C)

6745 a reçu le signal 2
(Contrôle-C)
$
```

Les signaux -Exemple 2

```
   #define _XOPEN_SOURCE
   #include <stdio.h>
   #include <stdlib.h>
   #include <unistd.h>
   #include <signal.h>

   void
gestionnaire (int numero_signal)
{
   int i;
   signal (numero_signal, gestionnaire);
   fprintf (stdout, "début du gestionnaire de signal %d \n",
                    numero_signal);
   for (i = 1; i < 4; i++) {
       fprintf (stdout, "%d\n", i);
       sleep (1);
   }
   fprintf (stdout, "fin du gestionnaire de signal %d\n", numero_signal);
}

   int
main (void)
{
   signal (SIGUSR1, gestionnaire);
   if (fork( ) = 0) {
       kill (getppid( ) , SIGUSR1);
       sleep (1);

         kill (getppid( ), SIGUSR1);
     } else {
        while (1) {
            pause( );
        }
     }
  return (0);
  }
```

Les signaux –Exemple 2 -Exécution

```
$ ./exemple_signal_3
début du gestionnaire de signal 10
1
début du gestionnaire de signal 10
1
2
3
fin du gestionnaire de signal 10
2
3
fin du gestionnaire de signal 10
  (Contrôle-C)
$
```

Les signaux –En complément

- Lectures complémentaires sur les signaux Unix/Linux

1. S. Moreaud, Communication par signaux. Cours de programmation système. IUT Bordeaux1 (France) .
2. Philippe MARQUET, Programmation des systèmes, Gestion des signaux, Lifl(France), 2005.
3. …etc

Outils de manipulation shell des outils IPC

- Vous pouvez examiner l'état de toutes les structures de communications interprocessus décrites dans cette partie — à l'exception des tubes — et connues par le système grâce à la commande ipcs de votre shell.

% ipcs

```
------ Shared Memory Segments --------
key        shmid   owner   perms   bytes    nattch   status
------ Semaphore Arrays ------
key        semid   owner   perms   nsems
------ Message Queues --------
key        msqid   owner   perms   used-bytes       messages
```

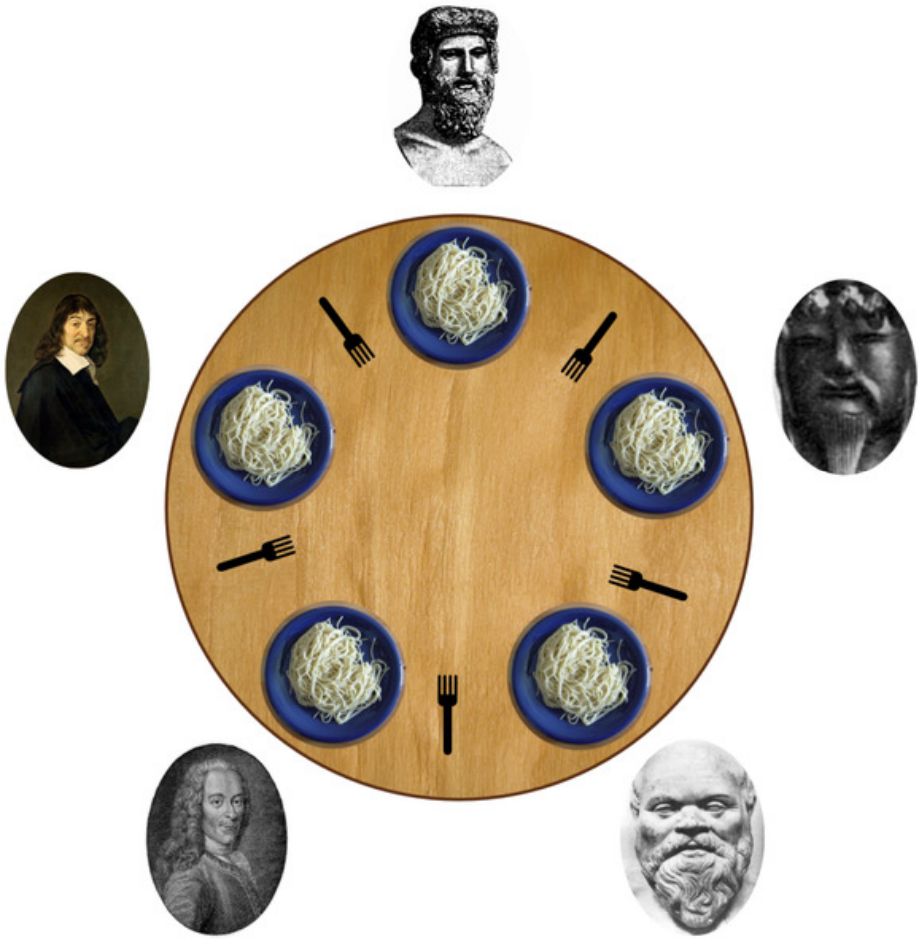
- Pour supprimer une structure de communication interprocessus (sémaphore, mémoire partagée, etc.) vous pouvez utiliser la commande shell
- % **ipcrm**

Questions ...?

ANNEXE

PROBLÈME DU DINER DES PHILOSOPHES

Le diner des philosophes (1/4)



Le diner des philosophes (2/4)

- **Description/Hypothèses :**
  - Init. On suppose cinq philosophes (possible d'avoir plus) se trouvent autour d'une table ronde;
  - chacun des philosophes a devant lui un plat de spaghetti ;
  - à gauche de chaque plat de spaghetti se trouve une fourchette.
  - Un philosophe n'a que trois états possibles :
    - penser pendant un temps indéterminé ;
    - être affamé (pendant un temps déterminé et fini sinon il y a famine) ;
    - manger pendant un temps déterminé et fini.

Le diner des philosophes (3/4)

- Des contraintes extérieures s'imposent à cette situation :
  - quand un philosophe a faim, il va se mettre dans l'état «affamé» et attendre que les fourchettes soient libres ;
  - pour manger, un philosophe a besoin de deux fourchettes : celle qui se trouve à gauche de sa propre assiette, et celle qui se trouve à droite (c-à-d les deux fourchettes qui entourent sa propre assiette);
  - si un philosophe n'arrive pas à s'emparer d'une fourchette, il reste affamé pendant un temps déterminé, en attendant de renouveler sa tentative.

La famine

- La famine est un problème que peut avoir un algorithme d'exclusion mutuelle.
- Il se produit lorsqu'un algorithme n'est pas équitable, c'est-à-dire qu'il ne garantit pas à tous les threads souhaitant accéder à une section critique une probabilité non nulle d'y parvenir en un temps fini.

Le diner des philosophes (4/4)

- ✓ Le problème consiste à trouver un ordonnancement des philosophes tels qu'ils puissent tous manger, chacun à leur tour.

Une fausse solution au problème

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

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                    /* philosopher is thinking */
        take_fork(i);                /* take left fork */
        take_fork((i+1) % N);        /* take right fork; % is modulo operator */
        eat( );                      /* yum-yum, spaghetti */
        put_fork(i);                 /* put left fork back on the table */
        put_fork((i+1) % N);         /* put right fork back on the table */
    }
}
```

Le diner des philosophes –Critique de la solution (1/3)

- **La procédure take_fork() attend que la fourchette spécifiée soit disponible et s'en saisit**
- **Cette solution ne fonctionne pas**
  - Supposons que les cinq philosophes prennent leurs fourchette gauche en même temps
  - Aucun ne pourra prendre sa fourchette droite
  - Il y aura un inter blocage

Le diner des philosophes –Critique de la solution (2/3)

- **Il est possible de modifier le programme de la façon suivante**
  - Après qu'un philosophe a pris sa fourchette gauche, le code détermine si sa fourchette droite est disponible
  - Si ce n'est pas le cas, le philosophe dépose sa fourchette gauche, attend pendant un certain temps, puis recommence le processus

Le diner des philosophes –Critique de la solution (3/3)

- **Cette proposition échoue également**
  - Tous les philosophes peuvent reprendre l'algorithme simultanément
  - Chacun prend sa fourchette gauche, puis, constate que sa fourchette droite n'est pas disponible, la repose et attend et ainsi de suite
  - Une telle situation, au cours de laquelle tous les programmes continuent de s' exécuter indéfiniment mais ne progressent jamais, se nomme privation de ressource (ou famine)

Solution au diner des philosophes

- **On utilise un tableau, appelé state , pour suivre l'état du philosophe: il mange, il pense, ou il a faim et tente de s'emparer des fourchettes**

- **Un philosophe ne peut passer à l'état manger que si aucun de ses voisins ne mange à ce moment la.**

- **On utilise un tableau de sémaphore, un par individu, de façon que les philosophe soient bloqués si les fourchettes sont prises**

Solution au diner des philosophes

```
#define N            5            /* number of philosophers */
#define LEFT         (i+N-1)%N    /* number of i's left neighbor */
#define RIGHT        (i+1)%N      /* number of i's right neighbor */
#define THINKING     0            /* philosopher is thinking */
#define HUNGRY       1            /* philosopher is trying to get forks */
#define EATING       2            /* philosopher is eating */
typedef int semaphore;            /* semaphores are a special kind of int */
int state[N];                     /* array to keep track of everyone's state */
semaphore mutex = 1;              /* mutual exclusion for critical regions */
semaphore s[N];                   /* one semaphore per philosopher */

void philosopher(int i)           /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {                /* repeat forever */
        think( );                 /* philosopher is thinking */
        take_forks(i);            /* acquire two forks or block */
        eat( );                   /* yum-yum, spaghetti */
        put_forks(i);             /* put both forks back on table */
    }
}
```

Solution au diner des philosophes

```
void take_forks(int i)                    /* i: philosopher number, from 0 to N-1 */
{
      down(&mutex);                       /* enter critical region */
      state[i] = HUNGRY;                  /* record fact that philosopher i is 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)                        /* i: philosopher number, from 0 to N-1 */
   {
         down(&mutex);                      /* enter critical region */
         state[i] = THINKING;               /* philosopher has finished eating */
         test(LEFT);                        /* see if left neighbor can now eat */
         test(RIGHT);                       /* see if right neighbor can now eat */
         up(&mutex);                        /* exit critical region */
   }

   void test(i)                             /* i: philosopher number, from 0 to N-1 */
   {
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
    }
   }
```