جامعة ابن طفيل Université Ibn Tofail
ⵜⴰⵙⴷⴰⵡⵉⵜ ⵅⵓ ⵟⵓⴼⴰⵢⵍ Ibn Tofail University

Système d'Exploitation -Synchronisation-

Med. AMNAI
Filière SMI-S4
Département d'Informatique

2023-2024

Plan

Plan

Plan

Plan

Plan

Classes d'Interruptions

- Une interruption est la suspension de l'exécution normale d'un processus pour exécuter un autre processus ;
- La commutation du processeur, d'un processus à un autre, est provoquée par un signal ;
- Différentes classes d'interruption
  - Les E/S ;
  - Les exceptions (instructions illégales, division par zéro, instruction inconue,...) ;
  - Synchronisation via l'horloge ;
  - Défauts matériels.

Situation de compétition (Race condition)

- Situation où plusieurs processus accèdent à la même ressource, de manière concurrente (au même moment , souvent la mémoire) ;
- Faille qui fait que le résultat va dépendre de l'ordre d'accès à la ressource par les différents processus ;
- **Question** : Que se passe-t-il si un processus est interrompu lors de son accès à la ressource, **laissant la main** à un autre processus qui utilise la même ressource ?
- **Solution** : Il faut un mécanisme d'**exclusion mutuelle** pour contrôler l'accès à la ressource

Section Critique

- Une suite d'instructions d'un programme accédant à une ressource partagée est appelée une section critique (SC) ;
- Critères requis pour bien gérer les race conditions :
  1. À tout moment, au plus un processus en SC ;
  2. Aucune hypothèse sur la vitesse et le nombre de CPU ;
  3. Aucun processus s'exécutant en dehors d'une SC ne doit bloquer les autres ;
  4. Aucun processus ne doit attendre indéfiniment avant de pouvoir entrer dans une SC (éviter les deadlocks et la famine).

Solutions d'Exclusion Mutuelle

Plusieurs solutions sont envisageables pour réaliser l'exclusion mutuelle :

1. Masquage des interruptions
2. Variables de verrouillage
3. Alternance
4. Algorithme de PETERSON
5. Instruction TSL (Test and Set Lock)
6. Primitives SLEEP & WAKEUP
7. Sémaphores
8. Diner des Philosophes.

Masquage d'Interruptions

- Le processus **masque** les interruptions, **avant d'entrer** dans une section critique (**SC**) ;
- Il ne peut être alors suspendu durant l'exécution de la section critique ;
- **Réactivées** les interruptions à la **sortie de la SC** ;
- **Inconvénions**
  - Si le processus **ne restaure pas** les interruptions à la sortie ;
  - La solution n'assure **pas l'exclusion mutuelle**, si le système n'est pas **monoprocesseur** ;
  - Les autres **processus** (cas **multiprocesseur**) exécutés par un autre processeur pourront **accéder aux objets partagés** ;
- Néanmoins, ça peut être pratique pour le kernel (**mise à jour**).

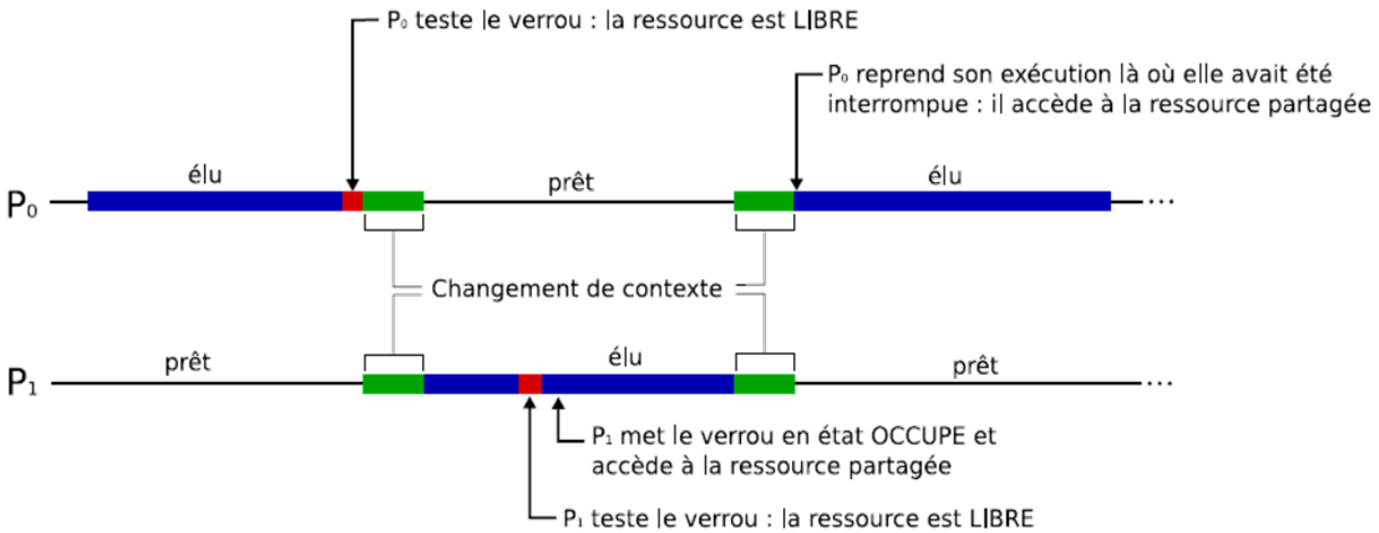
Variables de Verrouillage (Verrou)

1. Lock variables ;
2. Principe :
   - Associer à chaque ressource partagée une variable « **verrou** » prenant la valeur **LIBRE** ou **OCCUPE** ;
   - Ce verrou est **consulté** et **modifié** par chaque processus pour pouvoir **accéder** à la ressource.

```
Verrou : variable unique partagee
Verrou <- LIBRE
Si verrou == LIBRE alors
    verrou <- OCCUPE
    section critique
    verrou <- LIBRE
Sinon (cas ou verrou == OCCUPE)
Attente active que verrou passe a LIBRE
```

Inconvénions

- On déplace le problème sur le verrou : le **verrou** devient la ressource **partagée** ;
- Si l'ordonnanceur est préemptif le processus peut être **interrompu** entre le **test** du verrou et l'**accès** à la ressource.
- Donc Nous avons **deux processus en section critique**.



Alternance Stricte

Variable **tour** mémorise le tour du processus qui peut entrer en section critique :

```
Processus A

répéter{
    while tour <> 0 attendre
    section critique
    tour <- 1
    section non critique
}
```

```
Processus B

répéter{
        attendre while tour <> 1 attendre
        section critique
        tour <- 0
        section non critique
}
```

Inconvénions

- Verrouillage avec attente active (spin lock) ;
- Viole la condition 3 de bonne gestion des race conditions ;
- Si un processus est **très long il peut bloquer un autre processus** pendant un certain temps.

Exclusion Mutuelle avec Drapeau

```
Processus A

Faire toujours
(1) partie neutre
(2) occup1 <- true
(3) Tant que occup2 attendre
    section critique
    occup1 <- false
```

```
Processus B

Faire toujours
(11) partie neutre
(22) occup2 <- true
(33) Tant que occup1 attendre
     section critique
     occup2 <- false
```

Inconvénion : Les deux processus A et B se bloquent mutuellement en (3) et (33) (deadlock)

Algorithme de PETERSON

- Utilisation de deux fonctions : **entrer_region()** ; **quitter_region()** ;
- Avant d'entrer dans la section critique appeler la fonction **entrer_region(proc)**.
- Si nécessaire, attendre (dans **entrer_region(proc)**) jusqu'à ce qu'il n'y ait plus de risque.
- A la fin de la section critique, le processus doit appeler **quitter_region()** pour indiquer qu'il quitte sa section critique et pour autoriser l'accès aux autres processus.

```
section non critique;
entrer_sc();

    section critique;

quitter_sc();
section non critique;
```

Algorithme de PETERSON

```
=>Processus 0 appelle entrer_SC()
    interesse[0] = TRUE
    tour = 0

=>Processus 1 appelle entrer_SC()
    interesse[1] = TRUE
    tour = 1
    Attente sur le while que processus 0
                        appelle quitter_SC()

=>Processus 0 et 1 appellent entrer_SC()
    interesse[0]=interesse[1] =TRUE
    tour = 0 puis tour = 1
    Donc processus 0 en SC et processus 1
                        bloque sur le while
```

```
int tour;
int interesse[2];   //Deux processu

voide entrer_sc(int proc){

    int autre;      //autre processus
    autre = 1 - proc;
    interesse[proc] = true;
    tour = proc ;   //on est interssé
    while(tour == proc && interesse[autre]==true);
}

voide quitter_sc(int proc){
    //Processus quite la section critique
    interesse[proc]=false;
}
```

RQ : L'algorithme de **PETRESON** a l'inconvénient de l'attente active (test répété sur une variable pour détecter l'apparition d'une valeur).

Test and Set Lock (TSL)

- Certains processeurs disposent d'une instruction permettant d'effectuer de manière indivisible (atomique) le **test** de la valeur d'un registre (ou en mémoire) et de lui **assigner** une nouvelle valeur.
- ==> Un processus ne peut plus être interrompu entre le **test** de la variable **verrou** et le **blocage du verrou**.

```
drapeau : variable partagee

entrer_sc(){
     tsl regestre, drapeau < == > registre <= drapeau
                                 drapeau < = 1
     cmp regitre, 0
     bne entrer_sc
     ret
}
quitter_sc(){
     mov drapeau, 0
     ret
}
```

RQ :

- Problème d'**attente active** implique gaspillage du temps CPU.
- Conditions 3 et 4 des race conditions pas toujours vérifiées.

Problème d'Attente Active

Comment éviter l'attente active ?

- **Primitives** :
  - **sleep**() : suspend le processus appelant jusqu'à ce qu'un autre processus vient le réveiller.
  - **wakeup**(processus) : réveille le processus passé en paramètre.
- **Idée** :
  - Quand un processus appelle **sleep**(), il s'endort.
  - Si un autre processus appelle **wakeup**(), il réveille le processus endormi.

Principe du Producteur & Consommateur

- **Ressource partagée** :
  - Un buffer (mémoire tampon) de taille BUFFER_SIZE.
- **Objectifs** :
  - Un ou pluiseurs producteurs produisent des objects ;
  - Chaque producteur qui vient de produire un objet l'ajoute au buffer ;
  - S'il est plein, le producteur s'endort en attendant une place libre ;
  - Un ou plusieurs consommateurs viennent retirer les objets ;
  - Si le buffer est vide, le consommateur s'endort.
- **Problème** :
  - Régler les race conditions sur l'accès au buffer.

Algorithme Producteur & Consommateur

```
#define N 100   //taille du buffer
int cpt=0       //nombre d'objets dans le buffer
producteur (){
    while TRUE{
        produire_objet
        if (cpt == N) sleep () //buffer plein

        mettre_objet

        cpt = cpt+1
        if (cpt==1)  //cpt était à 0
            wakeup (consommateur)
}}
Consommateur (){
    while TRUE{
        if (cpt == 0) sleep () //buffer vide

        retirer_objet

        Cpt=cpt-1
        if (cpt==N-1)           // avant cpt=N
            wakeup (producteur)
}}
```

Problème Producteur & Consommateur

- Conflit sur la variable **cpt**.
- **Illustration**
  - Un consommateur teste **cpt** et trouve **0** (sleep n'est pas encore exécutée).
  - L'ordonnanceur bascule sur un producteur.
  - Le producteur incrémente **cpt** et constate que le buffer était vide.
  - Le producteur lance un **wakeup**() perdu car le consommateur n'était pas encore endormi !
- **RQ** : Trouver une solution dont
  - prod /cons avancent à leur rythme sans que l'un bloque l'autre.
  - nombre de prod/cons ne doit pas être connu à l'avance et non fixe.
- **Solution**
  - Mémoriser le **wakeup**() et utiliser les **sémaphores**.

Principe des Sémaphores

Dijkstra a proposé de compter le nombre d'appels en attente d'une ressource partagée.

- Un sémaphore **S** est une structure constitué d'un **compteur** à valeurs entières qui mémorise le **nombre de réveils en attente** et d'une **file d'attente**.
- Un sémaphore sert à **bloquer** des processus en **attendant** qu'une **condition** soit réalisée pour leur réveil.
- Les processus **bloqués** sont placés **dans la file** d'attente.

Down & Up

Les sémaphores permettent de réaliser des **exclusions mutuelles** à l'aide des primitives **DOWN** et **UP**.

- **DOWN** :
  - **S <- S - 1**
  - **si S < 0** alors bloquer le processus dans la file (**S**)
- **UP**
  - **S <- S + 1**
  - **si S <= 0** alors débloquer le processus de la file (**S**)

```
        sémaphore mutex = 1
processus P1:                    processus P2:
DOWN (mutex)                     DOWN (mutex)
  section critique de P1           section critique de P2
UP (mutex)                       UP (mutex)
```

RQ : Un sémaphore ne peut être manipulé que par **DOWN** et **UP**.

Solution au Problème des Producteurs/Consommateurs

Solution au problème des Producteurs/Consommateurs avec deux **sémaphores** (full, empty) et un mutex protégeant le buffer (S.C.).

- **full**
  - **Compte** le nombre de **places** du buffer qui sont **occupées** ;
  - La liste contient les processus **consommateurs** bloqués ;
  - Initialisée à **0** avec une liste vide ;
- **empty**
  - **Compte** le nombre des **places** qui sont **vides** dans le buffer ;
  - La liste contient les processus **producteurs** bloqués ;
  - Initialisée à BUFFER_SIZE avec une liste vide ;
- **mutex**
  - Une sémaphore binaire initialisée à **1** ;

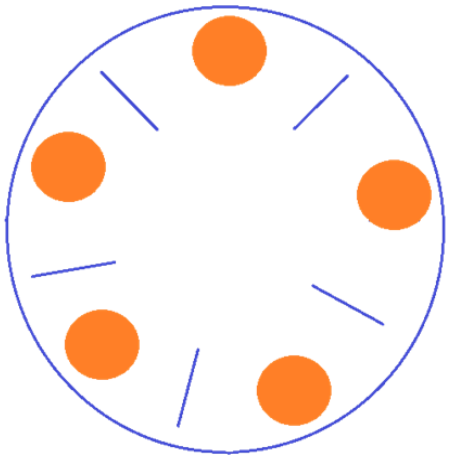
Solution au Problème des Producteurs/Consommateurs

```
#define N 100                   /*taille du buffer*/
typedef int semaphore           //défintion de type
semaphore mutex =1              //contrôle accès SC
semaphore libre = N             //Nombre places libres
semaphore plein = 0             // Nombre occupées

producteur (){
    while TRUE{
        produire_objet
        down(libre)             //décrémente nombre places libres
        down(mutex)             //entrer en SC
        mettre_objet
        up(mutex)               //sortie de la SC
        up(plein)               //incrémente nombre de place occupées
    }
}
Consommateur (){
    while TRUE{
        down(plein)             //décrémente nombre places occupées
        down(mutex)             //entrer en SC
        retirer_objet
        up(mutex)               //sortie de la SC
        up(libre)               //incrémente nombre de place libres
    }
}
```

Diner des Philosophes

- Cinq philosophes sont assis autour d'une table ;
- Chacun des philosophes a devant lui un plat de spaghetti ; à gauche de chaque assiette se trouve une fourchette ;
- **Pour manger** son plat de spaghettis, un philosophe a besoin de **deux fourchettes**.
- Un philosophe passe son temps à **manger** et à **penser**.



Diner des Philosophes

Un philosophe n'a que trois états possibles :

- **Penser** pendant un temps indéterminé (**Bloqué** ..E/S) ;
- Etre **affamé** (pendant un temps déterminé et fini sinon il y a famine) (**Prêt**) ;
- **Manger** pendant un temps déterminé et fini (**Actif**).

Diner des Philosophes

Des contraintes extérieures s'imposent à cette situation :

- Quand un philosophe a faim, il va se mettre dans l'état affamé et attendre (**Prêt**) 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 gauche de celle de son voisin (c'est-à-dire 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.

Contraintes Diner des Philosophes

- **Problème** : Si tous les philosophes **prennent en même temps chacun une fourchette**, aucun d'entre eux ne pourra prendre l'autre fourchette (situation d'**interblocage**) ? ?
- **Solution** : un philosophe ne prend jamais une seule fourchette. Les fourchettes sont les objets partagés. L'accès et l'utilisation d'une **fourchette** doit se faire en **exclusion mutuelle**.
- On utilisera le **sémaphore mutex** pour réaliser l'**exclusion mutuelle**.

Solution (1/3)

```
#define N 5

philosophe(i)
int i;
{
    while true{
        penser();
        prendre_fourchette(i);
        prendre_fourchette((i+1)%N);
        manger();
        poser_fourchette(i);
        poser_fourchette((i+1)%N);
    }
}
```

- **Inconvénion** : Risque de blocage si tous les philosophes, prennent une fourchette en même temps.

Solution (2/3)

```
#define N 5                 // Nombre de fourchettes
typedef int semaphore       //définition de type
semaphore mutex =1          //contrôle d accès S.C

philosophe(i)
int i;
{

    while true{
        penser();
        down (mutex);                   //Entrer en SC
        prendre_fourchette(i);
        prendre_fourchette((i+1)%N);//ex: sachant que N=5, si i==5 alors (i+1)%N=1
            manger();
        poser_fourchette(i);
        poser_fourchette((i+1)%N);
        up(mutex);                      //Sortie de la SC
    }
}
```

- **Inconvénion** : Un seul philosophe qui peut manger.

Solution (3/3)

```
#define N 5                 //nbre de philosophes
%Nefine GAUCHE (i-1)%N      //voisin gauche du philo i
#define DROITE (i+1)%N      //voisin droit du philo i
#define PENSE 0             // Philo pense
#define FAIM 1              //Philo veut fourchettes
#define MANGE 2             //Philo mange

typedef int semaphore;      //Def de type sémaphore (entier)
int etat[N];                //Etat de chaque philo init PENSE
semaphore mutex =1 ;        //contrôle d accès S.C
semaphore s[N];             //Un sémaphore par philo init 0

philosophe(i)
int i;
{
    while true{
        penser();
        prendre_fourchettes(i);
            manger();
        poser_fourchettes(i);
    }
}
```

```
prendre_fourchettes(i)
int i;
{
    down(mutex);//entrer en SC prise fr
    etat[i]=FAIM;//proc veut manger
    test(i);//prise de deux fourchettes
    up(mutex);//sortie de SC prise fr
    down(s[i]);//bloqué en file attente
}
poser_fourchettes(i)
int i;
{
    down(mutex);//entrer en SC poser fr
    etat[i]=PENSE;//etat M -> P
    test(GAUCHE);//réveiller phi gauche..
    test(DROITE);//réveiller phi droite..
    up(mutex);//sortie de SC poser fr
}
test(i)
int i
{
    if((etat[i]==FAIM)&&(etat[GAUCHE]!=MANGE)
                        &&(etat[DROITE]!=MANGE))
    {
        etat[i]=MANGE;//se préparer pour manger
        up(s[i]);//réveiller philo i par voisin.
    }
}
```