

# Contrôle de concurrence en mémoire commune

## Mécanismes primitifs d'exclusion mutuelle

# Contrôle de concurrence en mémoire commune

## Le contexte

- Systèmes centralisés

Architecture monoprocesseur : M processus préemptibles

Architecture multiprocesseurs à mémoire commune :

N processeurs supportent M processus s'exécutant en parallèle

- Plusieurs processus (thread) en exécution concurrente
  - l'ordre des accès à des mémoires partagées peut influencer le résultat final=>nécessité de synchronisation et de coopération
  - accès exclusif à une ressource partagée pour maintenir la cohérence des données

# Rappel Exclusion mutuelle

## Terminologie

- **ressource critique** ressource partagée dont l'utilisation n'est faite que par un processus à la fois
  - **exclusion mutuelle** condition de fonctionnement garantissant l'accès exclusif à la ressource
  - **section critique** phase d'utilisation d'une ressource critique par un processus
- Une section critique par ressource critique

# Rappel Exclusion mutuelle

## **Accès exclusif à une ressource**

### Hypothèses de fonctionnement

H1 Les vitesses relatives des processus sont quelconques

H2 Les priorités ou les droits des processus sont quelconques

H3 tout processus sort au bout d'un temps fini de sa section critique

# Rappel Exclusion mutuelle

## Spécification de comportement

- C1 Un processus au plus en section critique
- C2 Pas d'interblocage actif ou passif

Si aucun processus n'est en section critique et que plusieurs demandent à y entrer, alors un demandeur doit entrer en section critique au bout d'un temps fini

- C3 Un processus bloqué hors de sa section critique ne doit pas empêcher un autre d'y entrer
- C4 La solution doit être symétrique

# Rappel Exclusion mutuelle

Autres propriétés souhaitables

P1 Équité pas de famine de processus

P2 Respect des priorités des processus, des échéances

P1 et P2 contradictoires!

Remarque : l'exécution d'une section critique peut être vue comme une action atomique soit entièrement, soit pas du tout

# Exclusion mutuelle

## Schéma d'exécution des processus

```
While (true) {  
    entrée-SC // contrôle de l'accès de P  
    SC // utilisation  
    sortie-SC // P libère l'accès  
    hors-SC  
}
```

Remarque : on considère pour plus de généralité des processus cycliques

# Exclusion mutuelle par partage de mémoire

## **Le problème à résoudre**

Séquentialiser les sections critiques des processus concurrents

## **Le contexte matériel**

Architectures centralisées monoprocesseur ou multiprocesseur à mémoire commune

Accès atomique à un emplacement mémoire

# Exclusion mutuelle par partage de mémoire

## **Les mécanismes élémentaires fournis par le matériel**

- l'accès exclusif à un emplacement mémoire
- le masquage des interruptions
- les instructions spéciales type test and set

# Exclusion mutuelle par partage de mémoire

## Les types de solutions

- Exclusion mutuelle avec attente active

Les processus se détectent en conflit d'exclusion mutuelle en utilisant des mécanismes matériels, en cas d'attente ils exécutent une boucle

- Exclusion mutuelle avec attente passive

Libération du processeur en cas d'attente

Construction de mécanismes tels que :

sémaphores, moniteurs, objets protégés (Ada),  
méthodes synchronisées (Java)

# Exclusion mutuelle par attente active et variables communes

## Principe d'une solution

- définir un ensemble de variables partagées reflétant l'état d'occupation de la ressource et l'état des processus concurrents pour cette ressource
- avant d'entrer en section critique, un processus exécute un protocole testant et modifiant les variables d'état. Si la ressource n'est pas libre, il exécute une boucle d'attente.
- en sortant de section critique, un processus "libère" la ressource en modifiant les variables d'état

**Problème** : seule la lecture ou l'écriture d'une variable est atomique, entre deux instructions machines le processeur peut être alloué à un autre processus...

# Exclusion mutuelle par attente active et variables communes

```
//déclaration et initialisation des variables communes
Boolean M = false; // état d'occupation de la ressource

Tâche P0                                Tâche P1
While(true) {                            While(true) {
  //entrée en SC                          //entrée en SC
  while (M); //attente active              while (M); //attente active
  M=true; //ressource occupée              M=true; //ressource occupée
  Section_critique;                        Section_critique;
  //sortie de SC                            //sortie de SC
  M=false;                                  M=false;
  hors SC;                                  hors SC;
}
```

**La condition d'exclusion mutuelle n'est pas respectée!**

# Exclusion mutuelle par attente active et variables communes

Algorithmes de Dekker(1965) et de Peterson (1981)

Principe :

- 1- chaque processus annonce sa candidature à l'autre processus
- 2- en cas de candidatures simultanées, le conflit est réglé en donnant la priorité à un processus
- 3- pour une solution équitable la priorité doit être variable

Les solutions sont présentées pour 2 processus.

Généralisation difficile pour Dekker, possible pour Peterson

# Algorithme de Dekker

```
//déclaration et initialisation des variables communes
boolean C[2]={false,false}; //candidature
int T=0; //priorité
Tâche P0
while (true) {
  //entrée en SC
  C[0]=true;
  while (C[1]) { //P1 est en SC ou demande à y entrer
    while(T==1) C[0]=false; //P1 est prioritaire
    C[0]=true; //T=0 P0 recandidate
  }
  Section_critique;
  //sortie de SC
  C[0]=false;
  T=1;
  Hors SC;
}
```

# Algorithme de Dekker

```
Tâche P1
while (true) {
  //entrée en SC
  C[1]=true;
  while (C[0]) { //P1 est en SC ou demande à y entrer
    while(T==0) C[1]=false; //P1 est prioritaire
    C[1]=true; //T=0 P1 recandidate
  }
  Section_critique;
  //sortie de SC
  C[1]=false;
  T=0;
  Hors SC;
}
```

# Algorithme de Peterson

```
//déclaration et initialisation des variables communes  
boolean C[2]={false,false}; //candidature  
int T=0; //priorité
```

Tâche P0

```
while (true) {  
    //entrée en SC  
    C[0]=true; T=1;  
    while (C[1]and T==1); //attente en cas de conflit  
    Section_critique;  
    //sortie de SC  
    C[0]=false;  
    Hors SC;  
}
```

# Algorithme de Peterson

```
Tâche P1
while (true) {
  //entrée en SC
  C[1]=true;T=0;
  while (C[0]and T==0); //attente en cas de conflit
  Section_critique;
  //sortie de SC
  C[1]=false;
  Hors SC;
}
```

# Exclusion mutuelle par masquage des interruptions

## **Principe**

- avant d'entrer en section critique, un processus masque les interruptions
- en sortie de section critique, un processus démasque les interruptions

## **Remarques**

dans le cas d'une architecture monoprocesseur, la section critique est atomique

Le masquage et le démasquage des interruptions sont effectués en mode maître : appel système

# Exclusion mutuelle par masquage des interruptions

Tâche P

```
while (true) {  
    masquer_les_IT; // entrée en section critique  
    section_critique;  
    démasquer_les_IT; // sortie de section critique  
    hors_section_critique;  
}
```

## Critique de la solution

- entrées/sorties en SC : interblocage
- SC longue : perte d'interruptions
- dangereux un processus peut oublier de démasquer les IT
- inadaptée dans le cas multiprocesseur

## Conclusion

technique utile dans le noyau, mais pas au niveau utilisateur

# Exclusion mutuelle par instructions spéciales indivisibles

## Verrou et attente active

En général, le problème de l'exclusion mutuelle se résout en utilisant un verrou

```
while (true){  
    acquérir verrou;  
    Section critique;  
    relâcher verrou;  
    hors SC;  
}
```

# Exclusion mutuelle par instructions spéciales indivisibles

## Réalisation d'un verrou

Un verrou est une variable booléenne partagée

verrou=true la ressource critique n'est pas libre

verrou=false la ressource critique est libre

Le problème : acquisition atomique du verrou!

En général, il existe des instructions cablées telles que

Swap qui échange de manière atomique deux variables,

Test and Set qui teste et modifie de manière atomique une variable.

# Exclusion mutuelle par instructions spéciales indivisibles

## Réalisation d'un verrou avec swap atomique

```
swap( boolean x, y) {  
    boolean copie=x;x=y;y=copie;}  
boolean verrou=false; //variable partagée initialisée
```

Tâche P :

```
//entrée en section critique  
boolean macopie;  
while (true){  
    macopie=true;  
    do{ swap(macopie,verrou);}   
    while (not macopie); //attente active  
    section critique;  
    // sortie de section critique  
verrou=false;} Cours11 NFP137
```

# Exclusion mutuelle par instructions spéciales indivisibles

Réalisation d'un verrou avec swap atomique

- Les propriétés de l'exclusion mutuelle sont vérifiées sous réserve qu'un processus sorte au bout d'un temps fini de sa section critique

-Il peut y avoir famine de processus

-Interblocage possible si on tient compte des priorités

-La solution n'est pas adaptée au cas de multiprocesseur si un processus ne s'exécute pas sur un seul processeur.

Dans ce cas, il faut éviter qu'une interruption réquisitionne le processeur d'un processus autorisé à entrer en SC => masquage des interruptions

# Exclusion mutuelle par instructions spéciales indivisibles

## Réalisation d'un verrou avec swap cas multiprocesseur

Tâche P :

```
//entrée en section critique
boolean macopie;
while (true){
    macopie=true;
    while (true) {
        masquer les interruptions;
        swap(macopie,verrou);
        if(not macopie)break;//sortie si verrou acquis
        démasquer les interruptions;
    }
    section critique;
    // sortie de section critique
    verrou=false;
    démasquer les interruptions;
}
```

# Exclusion mutuelle par attente active

## Inconvénients

- mauvaise utilisation du processeur
- ralentissement des accès mémoires (congestion du bus par des accès répétés aux variables de synchronisation)
- difficulté de conception et complexité des algorithmes

## Cas d'utilisation

- sections critiques de courte durée
- réalisation de mécanismes sans attente active

## Conclusion

Construire des mécanismes sans attente active

# Les sémaphores

## Principe

- mécanisme de base permettant un auto-contrôle et le cas échéant une attente passive(blocage)
- contrôle de la concurrence basé sur un nombre d'autorisations d'accès :
  - si nombre d'autorisations  $\leq 0$  bloquer le demandeur
  - si nombre d'autorisations  $> 0$  accorder l'accès au demandeur
  - restitution d'autorisations réveil éventuel de demandeur(s) bloqué(s)

# Les sémaphores

## Définition (Dijkstra-1965)

Un sémaphore  $S$  est un objet partagé constitué de

- un entier  $E$  initialisé à une valeur  $\geq 0$
- une file d'attente  $F$  des processus bloqués

Un sémaphore est accessible uniquement par 3 primitives atomiques :

**P(S)** : Puis-je (Proberen)

contrôle d'autorisation + blocage éventuel du demandeur

**V(S)** : Vas-y (verhogen)

ajout d'une autorisation + déblocage éventuel d'un demandeur

**E0(S,I)** : initialisation du sémaphore à  $I \geq 0$  autorisations et file d'attente vide

# Les sémaphores

## Définition

Primitive P(sémaphore S) :

début

```
S.E=S.E-1;//retrait d'une autorisation
si S.E<0 alors bloquer le processus;
placer son id dans la file F;
```

fin

Primitive V(sémaphore S) :

début

```
S.E=S.E+1;//ajout d'une autorisation
si S.E≤0 alors //il y a au moins un processus bloqué
choix et retrait d'un processus de F;
réveil du processus;finsi;
```

fin

Primitive E0(sémaphore S, entier I) :

début

```
S.E=I; S.F=vide;
```

fin

# Les sémaphores

## Réalisation

- les primitives sont des sections critiques pour la ressource sémaphore : masquage des interruptions, instructions type test and set
- lors du blocage d'un processus, appel à l'allocateur qui élit un autre processus prêt
- lors du réveil d'un processus, appel à l'allocateur qui peut choisir le processus courant ou le processus réveillé

# Les sémaphores

## Programmation de l'exclusion mutuelle

Contexte commun

```
mutex : sémaphore; E0(mutex,1) // 1 seule autorisation
```

Tâche P :

```
while (true) {  
    P(mutex) ; // entrée en section critique  
    Section critique;  
    V(mutex) ; // sortie de section critique  
}
```

# Les sémaphores

## Contraintes d'utilisation

-respect des spécifications de programmation=>

pas d'entrée ou sortie de SC hors protocole

attention :exceptions, erreurs, trappes...en SC

-respect des spécifications de comportement : section critique de durée finie=>

pas de boucle infinie, ni de blocage en SC

pas de destruction de processus en SC

-éventuellement spécifier les règles d'attente : priorité, ancienneté, équité

# Les sémaphores

## Contraintes d'utilisation

Cas de plusieurs ressources critiques

Soient 2 processus qui se partagent 2 ressources critiques

On associe 2 sémaphores mutex1 et mutex2 à ces ressources

Tâche P0	Tâche P1
...	...
P(mutex1);	P(mutex2);
P(mutex2);	P(mutex1);
...	...
V(mutex1);	V(mutex1);
V(mutex2);	V(mutex2);
...	...

## Risque d'interblocage

# Les moniteurs

## Le concept de moniteur

(Hoare(1974), Brinch Hansen(1975))

Objectif : faciliter l'écriture de programmes corrects

Réalisation : les variables et procédures de synchronisation sont encapsulées dans un module

```
type m=moniteur
  début
  déclaration des variables locales;
  déclaration et corps des procédures du moniteur;
  // accessibles seulement en exclusion mutuelle
  fin
```

# Les moniteurs

Synchronisation par deux primitives :

**wait()** **signal()**

qui permettent de bloquer ou de réveiller un processus sur une condition

Condition : variable sans valeur implémentée par une file d'attente

Syntaxe et sémantique des primitives :

**Cond.Wait()** : bloque toujours le processus appelant

**Cond.Signal()** : réveille un processus bloqué dans la file d'attente associée à **Cond**

# Les moniteurs

## Exemple

```
moniteur cohorte
  condition groupe;
  entier compte =0;
  entier constante N=20;
procedure entrer;
debut
    si compte=N alors wait(groupe);
    compte=compte+1;
fin
procedure sortir;
debut
    compte=compte-1;
    si compte=N-1 alors signal(groupe)
fin
```

# Les moniteurs

## Exemple

```
tâche P
  while (true) {
    cohorte.entrer;
    suite d'actions;
    cohorte.sortir;
    hors groupe;
  }
```