

SUPPORT DE COURS
SYSTEMES D'EXPLOITATION
SYNCHRONISATION DE PROCESSUS

1. Définitions :

Synchronisation : La synchronisation introduit une condition d'attente sur un ou plusieurs processus parallèles.

Points de synchronisation : On impose un ordre de précedence dans le temps logique sur certains points de la trace temporelle des processus. On impose à certains processus une condition pour autoriser le franchissement de certains points de leur trace temporelle. Ces points sont appelés des points de synchronisation.

Définition (Andre, Herman et Verjus) : La synchronisation consiste à mettre en concordance plusieurs processus ayant atteint chacun un événement donné. Cette mise en concordance est partielle, les processus reprennent leur liberté jusqu'au prochain point de synchronisation.

Définition (Boksenbaum) : la synchronisation consiste à définir et à réaliser des mécanismes assurant le respect des contraintes liées à la progression des processus. La méthode consiste à : 1) définir à chaque processus ses points de synchronisation 2) associer à chaque point une condition de franchissement.

1. Les modèles de problèmes :

1.1 Section critique : Une séquence d'instructions manipulant des données partagées par plusieurs processus est appelée section critique car l'entrelacement des exécutions de ces instructions par différent processus peut conduire à des incohérences. La solution est que l'utilisation de cette section critique par un processus doit exclure toutes les autres processus : exclusion mutuelle.

1.2 Allocation de ressources : Dans une application parallèle (ou distribuée) les ressources mises à la disposition des processus sont en nombre limité. Les stratégies d'allocation de ressources sont :

- stratégie d'ordonnancement de requêtes
- stratégie de priorité et optimisation de nombre de requêtes satisfaites.

L'objectif est une utilisation optimale des ressources et une distribution équitable des ressources entre les processus demandeurs.

Deux solutions peuvent être envisagées :

- lors d'une requête de N ressources, le nombre de ressources est suffisant et la requête est satisfaite, alors que d'autres requêtes pour un nombre plus élevé de ressources restent en attente (risque de famine).
- lors d'une restitution de M ressources plusieurs processus demandeurs peuvent être servis.

Trois stratégies de solution peuvent être suivies :

- FIFO : les requêtes sont satisfaites suivant l'ordre d'arrivée
- Priorité aux petit-demandeurs : avec risque de famine des gros demandeurs.

- Ascenseur : les requêtes sont satisfaites selon le nombre croissant des ressources demandées. A chaque étape (étage) les requêtes en attente sont satisfaites si le nombre de ressources est suffisant. Si aucune requête n'est en attente on passe à l'étape (étage) suivant. L'allocation de ressources peut aboutir à un problème d'interblocage.

1.3 Modèle Producteur-Consommateur : C'est le modèle de coopération de processus, cette coopération se traduit par une communication de messages de processus producteurs vers des processus consommateurs.

L'échange de messages est géré par un système de communication. Généralement un producteur crée un message et demande au système de communication que le message soit transmis à un consommateur quelconque ou bien précis.

- dans la communication par boîte aux lettres d'Unix, les messages sont typés.
- le message peut être diffusé à un ensemble donné de consommateurs ou uniquement à ceux en attente au moment du dépôt.
- le langage ADA réalise la communication entre deux tâches par un rendez-vous entre ces tâches.

1.4 Modèle Lecteur-Rédacteur : Quand des processus partagent une structure de données (un fichier par exemple) qui peut être à un instant donné accédé par un processus lecteur qui ne fait que consulter ou par un processus rédacteur qui modifie. Les deux types de processus doivent respecter les contraintes suivantes :

- la lecture est simultanée : plusieurs lecteurs peuvent lire en même temps
- l'écriture est en exclusion mutuelle : un seul processus rédacteur à la fois peut accéder au fichier.
- Lorsque le fichier est libre, les lecteurs et rédacteurs suivent une stratégie de priorité :
 - variante1 : Priorité aux lecteurs tant que la ressource n'est pas occupé par un rédacteur.
 - variante2 : Priorité aux rédacteurs où une demande de lecture est acceptée si aucun rédacteur n'est en cours d'écriture ou en attente d'accès.
 - variante3 : FIFO mais en autorisant la lecture simultanée.
 - variante4 : Priorité aux lecteurs sans famine de rédacteurs où une demande de lecture est acceptée si la ressource est accédée en lecture et aucun rédacteur n'est en attente. A la libération de la ressource par un rédacteur, tous les lecteurs sont réveillés.

1.5 Modèle Client-Serveur : le modèle s'applique à toute activité distribuée (répartie). L'activité parallèle se structure en 2 processus : des processus systèmes qui offrent et exécutent des services et des processus utilisateurs qui émettent des requêtes de services. La relation entre ces processus est une relation de client à serveur.

Remarque : un processus serveur peut être lui-même client.

Le protocole client serveur est implémenté par l'appel procédural à distance (RPC : remote procedure call). L'implantation des RPC est difficile car le serveur n'est pas sur le même site que le client. Le passage des paramètres de l'appel est par valeur car toute transmission d'adresse est exclue.

Le problème de fiabilité du réseau de communication oblige des sémantiques du protocole de communication. Pour être sûre que la requête est bien reçue le client émet la requête à chaque délai de garde. Le serveur qui reçoit plusieurs versions d'une même requête peut envoyer plusieurs réponses. Certains protocoles assurent que le service a été exécuté au moins une fois, et d'autres assurent que le service a été exécuté une seule fois au plus.

2. Les outils de synchronisation :

2.1 Masquage des interruptions : Pour éviter l'entrelacement de l'exécution des instructions dans un environnement mono-processeur, il suffit de masquer les interruptions. Le risque est de perdre des informations véhiculées par ces interruptions. La solution est de masquer les

interruptions susceptibles de conduire des incohérences dans les données. UNIX utilise cette technique pour assurer la cohérence des tables qu'il administre.

2.2 Test-And-Set (TAS) : Dans un environnement multi-processeur, masquer les interruptions est insuffisant. Pour réaliser l'exclusion mutuelle une opération qui permettrait une lecture et une écriture sur une variable d'une manière indivisible est nécessaire.

Certains processeurs possèdent une instruction atomique TAS qui effectue la lecture d'un mot mémoire et écriture en k cycles en affectant le registre d'état. La valeur à écrire est souvent prédéfini. Cette instruction permet d'implanter un verrou au niveau Assembleur. D'autres processeurs possèdent une instruction d'échange atomique entre le contenu d'un registre et un mot mémoire tout en mettant à jour le registre d'état du processeur : XCH (exchange).

2.3 Les verrous : un verrou est une variable de type enregistrement :

```
type verrou=enregistrement {
    état : (ouvert, fermé) ;
    file : liste de processus en attente ;
}
```

Le verrou est manipulé par deux opérations atomiques (primitives) :

```
var V : verrou ;
```

<pre>Verrouiller(V) { if (V.état== ouvert) V.état=fermé ; else enfile(V.file) ; }</pre>	<pre>Deverrouiller(V) { if (¬filevide(V.file)) P=defiler(V.file) ; else V.état=ouvert; }</pre>
---	--

2.4 Les événements : Pendant l'exécution d'une application parallèle, différents événements peuvent se produire : expiration d'un délai, libération d'une ressource, envoi/réception d'un message. Certains systèmes possèdent le type événement et en général il est privé à un processeur.

Un événement est une variable de type enregistrement :

```
type événement : enregistrement {
    état : (déclenché non-déclenché) ;
    file : liste de processus en attente ;
}
```

L'événement est manipulé par 3 primitives :

- attendre() : bloque automatiquement les processus l'exécutant ;
- déclencher() : débloque un processus en attente de l'événement ;
- effacer() : permet à un processus de marquer un événement comme non-déclenché, si l'opération déclencher() ne le fait pas automatiquement (réinitialiser).

Exemple1 : un événement particulier utilisé dans les applications temps réels est l'événement « expiration de délai » (time-out). Deux primitives sont associées à cet événement :

- activer(N) : l'événement est déclenché après N unités de temps
- annuler() : pour annuler la dernière demande de délai du processus.

Exemple2 : UNIX (Linux) offre des primitives qui utilisent des signaux :

- pause() : bloque un processus jusqu'à réception d'un signal
- kill(processus-id, signal-id) : envoie un signal à un processus particulier
- alarm(n) : le signal SIGALARM est envoyé à un processus après n secondes.

Exemple3 : le noyau 68K KERNEL (systèmes temps réel) offre des primitives pour synchroniser des tâches sur des événements. Chaque tâche possède 16 événements (0..15).

```

-SignalEvent(Num-T :tache-id, Num-Ev :événement) ;
-WaitEvents(attendus : array[15] of Boolean, délai :integer, Var arrivés: array[15] of Boolean, Var
result: (OK, time-out));
- EventsOccured(attendus: array[15] of Boolean): Boolean;
-ClearEvents(evt: array[15] of Boolean);

```

2.5 Les sémaphores: C'est une variable de type enregistrement :

```

type sémaphore : enregistrement {
    e : entier
    file : liste de processus en attente
}

```

Le sémaphore est manipulé par deux primitives :

```
var S : sémaphore ;
```

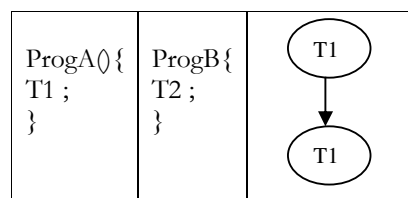
<pre> Procédure P(S) { S.e=S.e-1; if (S.e<0) enfiler(S.file); } </pre>	<pre> Procédure V(S) { S.e=S.e+1; if (S.e≤0) defiler(S.file); } </pre>
---	--

Généralement ils fonctionnent comme des distributeurs de tickets. Un processus demande un ticket en invoquant P(S) et s'il existe au moins un ticket (S.e>0) le processus le prends et continue son exécution sinon il est mis en attente dans une file d'attente associé au sémaphore (S.file). Un processus qui termine rend le ticket en invoquant V(S) et si la file d'attente n'est pas vide (S.e≤0), il réveille un processus en attente. Pour résoudre le problème de la section critique un sémaphore initialisé à 1, généralement appelé mutex, est utilisé.

Variantes de sémaphore :

- Sémaphore privé à un processus : seul ce processus effectue P (donc au plus un seul processus en attente) et V est effectué par tous les autres processus y compris le propriétaire.
- P immédiat : le processus qui demande un ticket ne sera pas bloqué dans le file d'attente s'il n'y'a pas de ticket, mais recevra un message d'échec et continue son exécution.
- P temporisé : le processus qui exécute P précise le délai maximum d'attente et se ce délai expire il sera réveillé et recevra un message d'échec.
- P avec priorité : les processus en attente seront enfilés selon leur priorités.
- Sémaphore binaire : l'entier S.e ne peut prendre que deux valeurs 0 ou 1 donc pas de file d'attente.
- P à plusieurs tickets: le processus peut demander n tickets en exécutant P (n est un argument de P) et libérer m tickets en exécutant V.

Exemple1 : Soient les deux programmes parallèles suivants. Ajouter les sémaphores pour garantir la relation de précédence.



Exemple2 : Soit une ressource partageable R qui peut être utilisée par au plus N processus simultanément. Ecrire l'algorithme d'un tel processus.

2.6 Le moniteur de Hoare (1974) :

Un moniteur décrit les règles de synchronisation nécessaires à une application parallèle donnée dans une unité syntaxique et sémantique unique. Par analogie, une classe en POO permet de construire une abstraction de données, et un moniteur permet de définir une abstraction de synchronisation. Le moniteur encapsule des données partagées et définit des méthodes d'accès à ces données. Le moniteur assure un accès en exclusion mutuelle. C'est le compilateur du langage de programmation parallèle qui assure l'exclusion mutuelle et quand un processus est en attente du moniteur il est ajouté à la mémoire tampon alloué au moniteur par le système.

Syntaxe : Une seule interface mais plusieurs implémentations :

```
monitor <nom-interface> defines
    <constantes, types et procédures exportés> ;
end <nom-interface> ;
monitor <nom-implémentation> implements
    <Implantation du moniteur> ;
end <non-implémentation>.
```

où d'une manière condensée (une seule implémentation) :

```
monitor <nom-moniteur> ;
    export <liste des identificateurs> ;
    <déclarations de constantes, types et variables>
    {procédures : méthodes d'accès}
end <nom-moniteur>.
```

Une procédure du moniteur peut éventuellement appeler une autre procédure du même moniteur. Ce genre de moniteur garantit l'exclusion mutuelle mais pas la synchronisation.

Les variables condition de Hoare : ce sont des classes d'objets de synchronisation dans un moniteur. Deux opérations complémentaires s'appliquent aux variables condition :

```
var c : variable condition ;
wait (c) : bloque toujours le processus exécutant cette opération et est enfilé dans une file d'attente associée à la variable condition.
signal(c) : débloque un processus de la file d'attente de c (si elle n'est pas vide).
```

Lors de l'exécution de signal(c) deux processus sont actifs à l'intérieur du moniteur : signaleur et le signalé. La règle de Hoare dit que le processus signaleur est suspendu donnant l'accès au signalé (prioritaire), mais pour éviter la famine du signaleur, il est prioritaire aux autres processus demandant l'accès au moniteur.

3. Solutions aux problèmes génériques :

3.1 Exclusion mutuelle : Pour palier au problème de la section critique.

L'exclusion mutuelle est garantie si les 4 conditions suivantes sont vérifiées simultanément :

- i) Exclusion : un seul processus en SC à la fois
- ii) Pas de blocage : si la SC est libre un processus en attente doit pouvoir y accéder (déroulement)
- iii) Attente limitée : un processus en attente d'une SC doit pouvoir y accéder au bout d'un temps fini
- iv) Équité : la solution est la même pour tous (pas de privilège).

Les processus ont cette forme avec i leur identifiant :

```
Proc(int i) {while (TRUE){
...

```

```
    Section d'entrée ;
```

```
    Section critique ;
```

```
Section de sortie ;
```

```
...  
}}
```

Solution1 : Masquage des interruptions

Section d'entrée : Masquer les interruptions
Section de sortie : effacer le masque

Inconvénients:

- i) la solution n'est valable que pour un environnement mono-processeur
- ii) perte d'informations véhiculées par les interruptions masquées
- iii) risque de famine

Solution2 : Attente active pour deux processus P0 et P1.

Algorithme1 : Une variable booléenne pour indiquer l'état de la section critique.

```
shared boolean libre=TRUE ;  
Proc(int i ) { while (TRUE){
```

```
    while (!libre);  
    libre:=FALSE;
```

```
Section critique;
```

```
    libre :=TRUE ;
```

```
}}
```

Algorithme2 : Une variable booléenne par processus pour indiquer leur état.

```
shared boolean demande[2] ;  
Proc(int i ) {while (TRUE){
```

```
    demande[i]:=TRUE;  
    while (demande[1-i]);
```

```
Section critique;
```

```
    demande[i]:=FALSE;
```

```
}}
```

Algorithme3 : Une variable entière pour indiquer le tour de chaque processus.

```
shared int tour ;  
Proc(int i) { while(TRUE) {
```

```
    while (tour!=i);
```

```
Section critique;
```

```
    tour :=1-i ;
```

```
}}
```

Question : les trois algorithmes ne garantissent pas l'exclusion mutuelle. Trouver quelle condition n'est pas vérifiée pour chacune des solutions.

Remarque : Les constantes TRUE et FALSE n'existent pas en C.

```
#ifndef FALSE  
#define FALSE 0  
#endif  
#ifndef TRUE  
#define TRUE (!FALSE)  
#endif
```

Algorithme de Peterson (pour 2 processus):

```

BOOLEAN flag[2]=FALSE ; int tour;
Process(int i){ While (TRUE) {

```

```

    flag[i]=TRUE;
    tour=1-i;
    While (flag[1-i] && tour==1-i) ;
    /*Section critique*/

```

```

    flag[i]=FALSE;
}}

```

Question : Vérifier que l'algorithme de Peterson assure l'exclusion mutuelle.

Algorithme du boulanger (pour n processus)

$\{(a,b)<(c,d) \text{ si } a<c \text{ ou si } a=c \text{ et } b<d\} \{ \max(a_0, a_1, \dots, a_{n-1})=k \text{ tel que } k \geq a_i \text{ pour } i=0..n-1 \}$

```

shared boolean choix[n]=False;
shared int tour[n]=0 ;

```

```

Process(int i) { while(True){

```

```

    choix[i]=True;
    tour[i]:=max(tour[0],...,tour[n-1])+1 ;
    choix[i]=False ;
    for( j=0 ; j<n ;j++) {
        while (choix[j]) ;
        while (tour[j] && (tour[j],j)<(tour[i],i));
    }

```

```

        Section critique() ;

```

```

        tour[i]=0 ;

```

```

    }}

```

Questions :

- A quoi servent les deux instructions $\text{choix}[i] = \text{True}$ et $\text{choix}[i] = \text{False}$?
- Pourquoi chaque processus commence par vérifier si $\text{choix}[j] = \text{vraie}$?
- Montrer que si un processus P_i est dans sa section critique et P_k ($k \neq i$) a calculé son $\text{tour}[k]$ alors $(\text{tour}[i], i) < (\text{tour}[k], k)$.
- Vérifier si dans le cas où plusieurs processus attendent que la section critique soit libérée alors le premier arrivé sera le premier servi.

Solution2 : les sémaphores

```

shared semaphore mutex :=1 ;
Proc(int i) {while(TRUE){
    P(mutex);
    Section critique;
    V(mutex);
}}

```

Solution3 : les verrous

```

shared verrou V :=ouvert ;
Proc(int i) {while (TRUE) {
    verrouiller(V);
    Section critique;
    deverrouiller(V);
}}

```

Question : Vérifier que les deux solutions vérifient l'exclusion mutuelle.

Solution4 : Donner la solution de l'EM basé sur les moniteurs

3.2 Producteur Consommateur

<pre>Producteur : prod(){ while(true){ produire(objet) mettre(objet, buffer) ; }} </pre>	<pre>Consommateur : cons(){while(true){ objet=retirer(buffer) ; consommer(objet) ; }} </pre>
---	--

Remarque : Le buffer est une ressource critique !

Solution 1 : Sémaphore

<pre>semaphore mutex=1 ; plein :=0 ; vide :=n ; Prod() {while(true){ produire(objet) ; P(vide) ; P(mutex) ; mettre (objet, buffer) ; V(mutex) ; V(plein) ; }} </pre>	<pre>Cons() {while(true){ P(plein) ; P(mutex) ; objet=retirer(buffer) ; V(mutex) ; V(vide) ; consommer(objet); }} </pre>
---	--

Question : vérifier que les contraintes du modèle sont respectées.

Solution2 : Moniteur

<pre>monitor Prod-Cons ; export mettre(objet), retirer(objet); const N :=50 ; var vide, plein : condition ; cpt : integer ; buffer : array[50] of objets ; procedure mettre (objet) ; begin if cpt=N then wait(plein) endif buffer[cpt]=objet; cpt:=cpt+1; if cpt=1 then signal(vide) endif end mettre; </pre>	<pre>procedure retirer(objet); begin if cpt=0 then wait(vide) endif objet:=buffer[cpt]; cpt:=cpt-1; if cpt=N-1 then signal(plein) endif end retirer; begin cpt:=0; end Prod-Cons; </pre>
---	---

Les processus demandeurs:

<pre>Producteur : begin produire(objet); Prod-Cons.mettre(objet) ; End </pre>	<pre>Consommateur : begin Prod-Cons.retirer(objet) ; consommer(objet) ; end </pre>
---	--

Questions :

- 1) pourquoi l'opération produire et consommer sont en dehors du moniteur
- 2) peut-on utiliser une seule variable condition et pourquoi ?
- 3) vérifier que la solution respecte les contraintes du modèle.

3.3 Modèle lecteur-rédacteur :

Solution1 : Moniteur de Hoare

Interface :

```
monitor Lect-Redact defines
```



```

    procedure demander-lecture ;
    procedure terminer-lecture ;
    procedure demander-ecriture ;
    procedure terminer-ecriture ;
end Lect-Redact ;

```

Implémentation : suivant les variantes du modèle

<pre> monitor variante implements Lect- Redact ; var n-lect :integer ; redaction : boolean ; lecture, ecriture : condition ; procedure demander-lecture ; begin if redaction then wait(lecture) endif n-lect:=n-lect+1; signal(lecture); end demander-lecture ; procedure terminer-lecture ; begin n-lect :=n-lect-1 ; if n-lect =0 then signal(ecriture) endif end terminer-lecture; </pre>	<pre> procedure demander-ecriture; begin if n-lect>0 or ecriture then wait(ecriture) endif redaction=true; end demander-ecriture; procedure terminer-ecriture; begin redaction=false; <u>signal(lecture);</u> <u>if n-lect=0 then signal(ecriture)</u> <u>endif</u> end terminer-ecriture; begin redaction=false; n-lect:=0; end variante. </pre>
--	--

Processus utilisateurs :

<pre> Lecteur: begin variante.demander-lecture; lire() ; variante.terminer-lecture ; end </pre>	<pre> Redacteur : begin variante.demander-ecriture ; ecrire() ; variante.terminer-ecriture ; end </pre>
--	--

Questions :

- 1) quelle variante est implémentée par cette solution ?
- 2) vérifier que les contraintes du modèle sont respectées
- 3) pourquoi y'a-t-il un signal(lecture) avant un signal(ecriture) dans la procedure terminer-ecriture?

Solution2 : Sémaphore

<pre> semaphore mutex :=1, ecriture=1 ; int n-lect :=0 ; Redact () {while(true){ P(ecriture); ecrire() ; V(ecriture); }} </pre>	<pre> Lect () {while(true){ P(mutex) n-lect++; if (n-lect==1) P(ecriture); V(mutex); lire() ; P(mutex); n-lect-- ; if (!n-lect) V(ecriture) ; V(mutex); }} </pre>
--	--

Questions :

- 1) Quelle variante est implémentée par cette solution ?

2) pourquoi l'opération écrire() est entre un P puis V alors que l'opération lire() est entre un V puis P ?

Exercices :

Problème de la voir unique :

Soit un pont, à voie unique, traversé par des véhicules en sens inverse (sens A et sens B). A tout moment, le pont ne doit contenir que des véhicules allant dans un sens uniquement. On assimilera les véhicules à des processus parallèles.

Voiture AversB :

Demande d'accès AB ;

Circulation sur le pont de A vers B;

Libère le pont;

Voiture BversA :

Demande d'accès BA ;

Circulation sur le pont de B vers A;

Libère le pont;

1. Parmi les modèles étudiés en classe (producteur/consommateur, lecteur/rédacteur, les philosophes), ce problème correspond à quel modèle ?
2. Proposer une solution en utilisant les sémaphores (les codes de demandes d'accès et de sorties) de façon à ce que les processus respectent les règles de circulation, si 1) le nombre de véhicules sur le pont est 1 2) le nombre de véhicules le pont dans un sens est illimité 3) si le nombre est limité

Problème du carrefour :

La circulation à un carrefour à deux voies est réglée par des feux verts ou rouges. Les règles de circulations sont les suivantes :

- quand le feu est rouge sur une voie, les voitures y circulant doivent attendre ; quand le feu est vert les voitures peuvent traverser le carrefour,
- une voiture se présentant au carrefour doit le franchir au bout d'un temps fini,
- les feux de chaque voie passent alternativement du rouge au vert,
- à un instant donné, le carrefour ne doit contenir que des voitures d'une même voie,
- les voitures traversent le carrefour en ligne droite.

Ecrire les programmes correspondant aux processus :

- changement assurant la gestion de la commande des feux,
- traversée1 et traversée2 assurant respectivement la traversée d'une voiture des voies 1 et 2, quand le carrefour peut contenir une seule voiture à la fois, puis au plus k voitures à la fois.

Mme YAICI.