Université des Sciences et de la Technologie
Houari Boumediene
USTHB

Parallélisme et synchronisation de processus

Dr BOUYAKOUB F. M
bouyakoub.f.m@gmail.com

Octobre 2012

Plan du cours

- Notion de parallélisme:

  ➔problèmes de la programmation concurrentielle

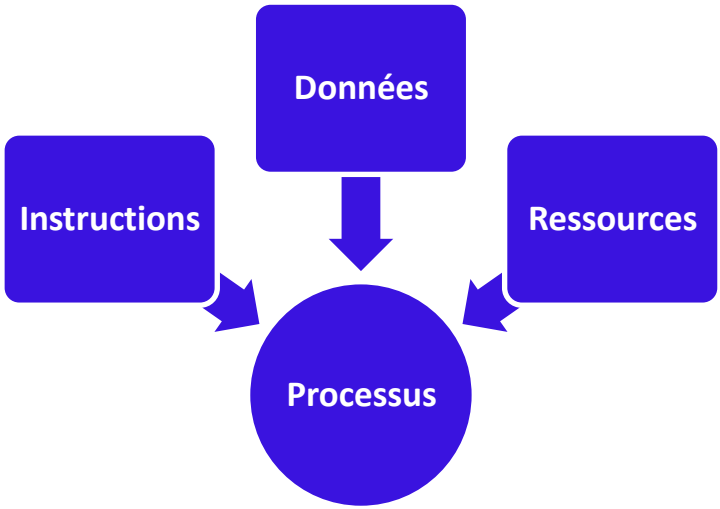
- Synchronisation de processus:
  - Le problème de l'exclusion mutuelle
  - Mécanismes d'exclusion mutuelle

Notion de parallélisme

bouyakoub.f.m@gmail.com

Définition d'un processus

- Un processus est l'instance d'un programme en cours d'exécution à un instant T et son environnement d'exécution.
- Un processus est défini par:
  - Un environnement processeur
  - Un environnement mémoire



Scénarios d'exécution de processus

- Considérons deux programmes distincts P et Q et leurs processus associés p et q.

Notion de tâche ou *thread* (processus léger)

- Une tâche est une unité élémentaire de traitement ayant une cohérence logique.

Exemple:

P1

```
Lire (X);
X=X+1;
Ecrire (X);
```

Programmes concurrents

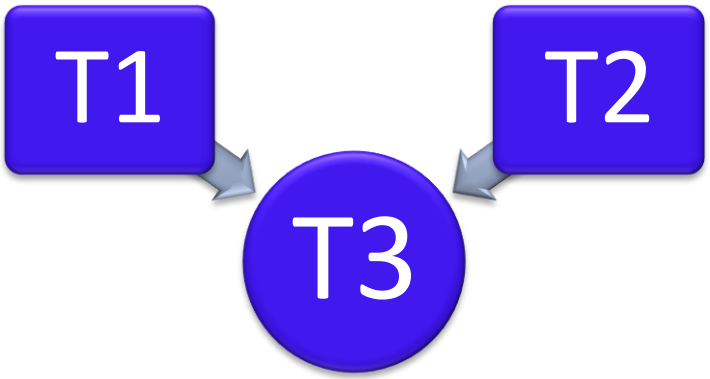
- Un programme concurrent est un programme dont certaines tâches peuvent s'exécuter en parallèle, mais partagent les mêmes variables.
  - Exemple:

E=(X-Y)/(Z+T) ➔

T1➔ X-Y

T2 ➔ Z+T

T3➔ T1/T2



Système de tâches

- La concurrence implique deux modes d'exécution: le parallélisme et la séquentialité des tâches ➔ relation de précédence "<" entre les tâches.
  - Exemples:
    - T1 < T2 ➔ la terminaison de T1 doit avoir lieu avant le début de T2
    - T1 ~~<~~ T2 et T2 ~~<~~ T1 alors l'ordre d'exécution de T1 et T2 est quelconque ➔ T1 et T2 sont en parallèle

Exemple de tâches et parallélisme

Programme P

```
Lire(A);
Lire(B);
B=B*2;
C=A*B;
Afficher(C);
```

La synchronisation de processus

bouyakoub.f.m@gmail.com

Problème du parallélisme

- Les processus multiples dans un SE n'évoluent pas toujours de manière indépendante. Ils coopèrent pour réaliser des tâches communes, ils partagent donc des données et des ressources.

➔ pour mettre en œuvre cette opération, des mécanismes de synchronisation doivent être offerts par le SE pour ordonner l'exécution de ces processus.

Partage de ressources et de données

- L'exécution d'un processus nécessite des ressources physiques (mémoire, CPU, périphérique…) ou logiques(variable, fichier…).
  - Des ressources peuvent être utilisées par plusieurs processus en même temps ➔ ressources partageables
  - Des ressources ne peuvent être utilisées que par un processus à la fois ➔ à un seul point d'accès ou à accès exclusif ou non partageable.

  ➔ prévoir des mécanismes pour ordonner l'accès à ces ressources

Exemple 1: problèmes des variables partagées

Programme de réservation:

```
Si (PlaceDispo>0)
Alors PlaceDispo--;
        Afficher ("place réservée");
Sinon Afficher ("Plus de place");
```

Supposons qu'on a plusieurs demandes de réservation émanant d'agences différentes ➔ plusieurs exécutions simultanées du même code avec la variable partagée *PlaceDispo*

Scénario possible

- 2 requêtes → 2 processus P1 et P2 et *PlaceDispo*=1

P1	P2
Si (PlaceDispo>0) Alors PlaceDispo--; Afficher ("place réservée"); Sinon Afficher ("Plus de place");	Si (PlaceDispo>0) Alors PlaceDispo--; Afficher ("place réservée"); Sinon Afficher ("Plus de place");

➔P1 et P2 réserveront la même place (incohérence)

Exemple 2: cas d'un *spool* d'impression

Quand un processus veut imprimer un fichier, il doit placer le nom de ce fichier dans un file d'attente. Un processus démon, le spooler, vérifie périodiquement s'il y a des fichiers à imprimer. Si c'est le cas, il les imprime et retire leur nom de la file.

Scénario d'exécution

Supposons que :

- La file d'impression ait un très grand nombre d'emplacements, numérotés 0,1,2,...,N.
- Chaque emplacement peut contenir le nom d'un fichier à imprimer.
- Tous les processus partagent la variable "**suivant**" qui pointe sur le prochain emplacement libre de la liste.
- Deux processus A et B veulent placer chacun un fichier dans la file d'impression et que la valeur de "**suivant**" est 7.
- Le processus A place son fichier à la position "**suivant**" qui vaut 7. Une interruption d'horloge arrive immédiatement après et le processus A est suspendu pour laisser place au processus B.
- Ce dernier place également son fichier à la position "**suivant**" qui vaut toujours 7 et met à jour "**suivant**" qui prend la valeur 8. Il efface ainsi le nom du fichier placé par le processus A.
- Lorsque le processus A sera relancé (restauration du contexte), il met à jour la valeur de "**suivant**" qui prend la valeur 9.

- Sous ces conditions, deux problèmes se présentent:
  - Le fichier du processus A ne sera jamais imprimé;
  - **"Suivant"** ne pointe plus sur le prochain emplacement libre.

solution

- Synchroniser les processus A et B
- Lorsqu'un processus accède à la variable "**suivant**", les autres processus ne doivent ni la lire, ni la modifier jusqu'à ce que le processus ait terminé de la mettre à jour.
- D'une manière générale, il faut empêcher les autres processus d'accéder à un objet à un seul point d'accès si cet objet est en train d'être utilisé par un processus ➔ **exclusion mutuelle**.
- Les situations de ce type, où deux ou plusieurs processus lisent ou écrivent des données partagées et où le résultat dépend de l'ordonnancement des processus, sont qualifiées d'**accès concurrents**.

L'exclusion mutuelle

bouyakoub.f.m@gmail.com

L'*exclusion mutuelle*

Le problème de conflit d'accès serait résolu, si on pouvait assurer que deux processus ne soient jamais en **section critique** en même temps au moyen de l'***exclusion mutuelle***.

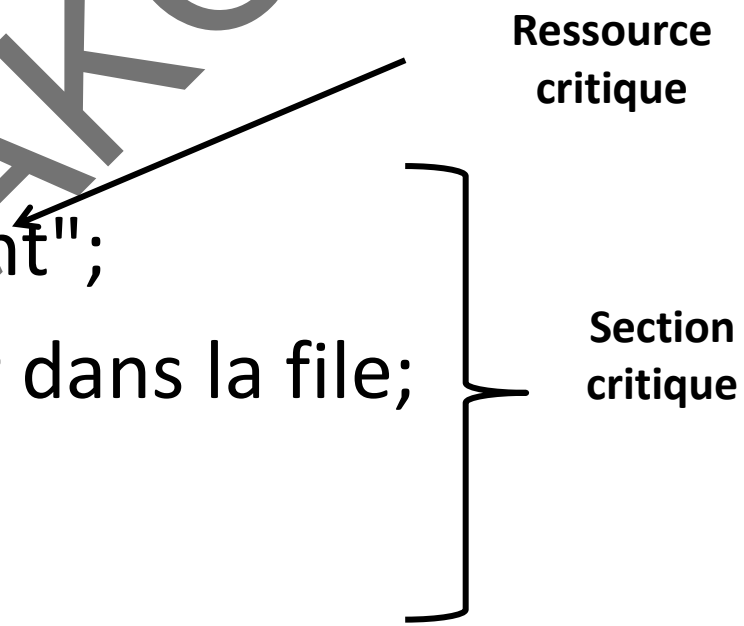
C'est quoi une section critique?

Ressource critique et section critique

Ressource critique: une ressource critique est un objet qui ne peut être accédé simultanément par plusieurs processus (imprimantes, fichiers, variables…).

Section critique: Ensemble de suites d'instructions qui opèrent sur un ou plusieurs objets critiques et qui peuvent produire des résultats imprévisibles lorsqu'elles sont exécutées simultanément par des processus différents.

Exemple du *spool* d'impression



- Lecture de la variable "suivant";
- L'insertion du nom de fichier dans la file;
- La mise à jour de "suivant";

Structure des processus en section critique

Début programme

<Section non critique>;

<Demande d'entrée en section critique>;

<Section critique>;

<Demande de sortie de la section critique>;

<Section non critique>;

Les 4 propriétés de *Dijkstra* pour l'exclusion mutuelle

- Un seul processus en SC;
- Aucun processus s'exécutant à l'extérieur de la SC ne doit bloquer d'autres processus;
- Un processus désirant entrer en SC y entre au bout d'un temps fini; pas de privation d'y entrer vis-à-vis d'un processus.
- La solution doit être la même pour tous les processus: aucun processus ne doit être privilégié.

Conditions de réalisation de l'exclusion mutuelle

L'exclusion mutuelle n'est pas garantie si:

1. Un processus peut entrer en SC alors qu'un autre s'y trouve déjà;
2. Un processus désirant entrer en SC ne peut pas y entrer alors qu'il n'y a aucun processus en SC (**progression**);
3. Un processus désirant entrer en SC n'y entrera jamais car il ne sera jamais sélectionné lorsqu'il est en concurrence avec d'autres processus (**attente bornée**);
4. Un processus en SC ne sort jamais de la SC.

Mécanismes d'exclusion mutuelle

Mécanismes utilisés pour l'exclusion mutuelle

Mécanismes d'exclusion mutuelle

Solution logicielle

- Variables de verrouillages
- L'alternance
- Solution de *Peterson*

Solution matérielle

- Masquage des interruptions (monoprocesseur)
- Instruction indivisible (multi-processeur)

Mécanismes utilisés pour l'exclusion mutuelle

Mécanismes d'exclusion mutuelle

Solution logicielle

- Variables de verrouillages
- L'alternance
- Solution de *Peterson*

Solution matérielle

- Masquage des interruptions (monoprocesseur)
- Instruction indivisible (multiprocesseur)

Variables de verrouillages

- Utilisation d'une variable de verrouillage partagée **verrou**, unique, initialisée à 0.
- Pour rentrer en SC, un processus doit tester la valeur du **verrou**.
- Si elle est égale à 0, le processus modifie la valeur du **verrou ← 1** et exécute sa SC.
- À la fin de la section critique, il remet **verrou ← 0**.

Algorithme 1

Algorithm 1 Verrouillage

```
while verrou ≠ 0 do
    ; / / Attente active
end while
verrou ← 1
Section_critique() ;
verrou ← 0
```

Scénario possible:

- Un processus est suspendu juste après avoir lu la valeur du verrou qui est égal à 0.
- Ensuite, un autre processus est élu. Ce dernier teste le verrou qui est toujours égal à 0, met verrou ←1 entre dans sa SC.
- Ce processus est suspendu avant de quitter la SC.
- Le premier processus est alors réactivé, il entre dans sa SC et met verrou ←1.

➔ Les deux processus sont en même temps en SC.

Mécanismes utilisés pour l'exclusion mutuelle

Mécanismes d'exclusion mutuelle

Solution logicielle

- Variables de verrouillages
- L'alternance
- Solution de *Peterson*

Solution matérielle

- Masquage des interruptions (monoprocesseur)
- Instruction indivisible (multiprocesseur)

L'alternance: algorithme 1

- L'alternance consiste à utiliser une variable tour qui mémorise le tour du processus qui doit entrer en SC.

Processus P_i	Processus P_j

while (1)
{
while (tour !=i) ; // attente active
<SectionCritique> ;
tour = j ;
Suite processus ;
…
}
	while (1)
	{
	while (tour !=j) ; // attente active
	<SectionCritique>;
	tour = i;
	Suite processus ;
	…
	}

Analyse de l'approche

- L'exclusion mutuelle est assurée car l'entrée en SC est déterminée par la valeur de tour qui indique l'identité du processus qui accédera en SC.

- Problème: Il est possible qu'un des deux processus ait plus souvent besoin d'entrer en SC que l'autre; l'algorithme lui fera attendre son tour bien que la SC ne soit pas utilisée.

  ➔Un processus peut être bloqué par un processus qui n'est pas en section critique.

L'alternance: algorithme 2

- On remplace la variable commune tour par un tableau qui indique l'état du processus par rapport à la SC (demandeur ou non).

Processus P_i	Processus P_j

while (1)
{
Demande[i]= true;
while (Demande[j]) ; // attente active
<SectionCritique> ;
Demande[i]=false
Suite processus ;
…
}
	while (1)
	{
	Demande[j]= true;
	while (Demande[i]) ; // attente active
	<SectionCritique> ;
	Demande[j]=false
	Suite processus ;
	…
	}

Problème de l'algorithme 2

- Dans le cas où les deux processus exécutent en même temps les instructions **Demande[i]= true** et **Demande[j]= true**

➔**blocage mutuel entre les deux processus**

Mécanismes utilisés pour l'exclusion mutuelle

Mécanismes d'exclusion mutuelle

Solution logicielle

- Variables de verrouillages
- L'alternance
- Solution de *Peterson*

Solution matérielle

- Masquage des interruptions (monoprocesseur)
- Instruction indivisible (multiprocesseur)

Algorithme de *Peterson*

- La première solution correcte à l'exclusion mutuelle, dans le cas de deux processus, a été introduite par *Dekker* (1965) et simplifiée par *Peterson* (1981).

- Le principe: Chaque processus indique dans une variable **Demande[x]** son souhait d'entrer en section critique; une variable commune, **tour**, évite les attentes infinies à l'entrée de la section critique.

Algorithme de *Peterson*

Processus P_i

```
while (1)
{
Demande[i]= true;
Tour =i;
while ((Demande[j]==true) && (tour ==i)); // attente active
<SectionCritique> ;
Demande[i]=false ;
Suite processus ;
...
}
```

A faire pour la semaine prochaine!

Montrer que l'algorithme de ***Peterson*** vérifie les propriétés de l'exclusion mutuelle.

Inconvénient de ces solutions: L'attente active

- Les processus souhaitant accéder à la SC exécutent une boucle en attendant l'accès à la SC ➔ **consommation de ressources CPU**
- Soient deux processus **H** (priorité haute) et **B** (priorité basse):

  Si **B** est en SC et **H** termine une E/S et veut accéder en SC ➔ **H** se mettra en attente active
  - Avec les règles de *scheduling* **H** va boucler sans fin et **B** ne pourra pas sortir de la SC.

Mécanismes utilisés pour l'exclusion mutuelle

Mécanismes d'exclusion mutuelle

Solution logicielle

- Variables de verrouillages
- L'alternance
- Solution de *Peterson*

Solution matérielle

- Masquage des interruptions (monoprocesseur)
- Instruction indivisible (multiprocesseur)

Masquage des interruptions

- Dans un système monoprocesseur, et en l'absence d'interruption, un processus s'exécute de façon continue, instruction après instruction

  ➔ En l'absence d'interruption, le problème de l'EM pour l'accès aux données n'existe pas.

- Une façon simple et brutale d'assurer l'indivisibilité d'une suite d'instructions consiste à masquer les interruptions au début de la section critique et à les démasquer à la fin.

Remarques

1. Le masquage ou le démasquage des interruptions ne peuvent se faire que si l'ordinateur s'exécute en mode maître. Dans un processus utilisateur qui s'exécute en mode esclave, il faut impérativement appeler les séquences à masquer par des appels système permettant le passage en mode maître.
2. Pour éviter la perte de signaux d'interruptions, les séquences masquées ne peuvent pas être trop longues.

Schéma décrivant le masquage d'It pour l'EM

while(1)
{
< …….. >
Masquer les interruptions ;
< Section Critique >
Démasquer les interruptions ;
<……..>
}

Mécanismes utilisés pour l'exclusion mutuelle

Mécanismes d'exclusion mutuelle

Solution logicielle

- Variables de verrouillages
- L'alternance
- Solution de *Peterson*

Solution matérielle

- Masquage des interruptions (monoprocesseur)
- Instruction indivisible (multiprocesseur)

L'instruction TSL (*Test and Set Lock*)

- La plupart des processeurs ont une instruction atomique ayant pour paramètres un registre et une adresse mémoire (partagée par tous les processeurs) appelée **drapeau**.
- L'instruction TSL garantit son exécution en un seul cycle indivisible; elle charge, dans un registre, le contenu du drapeau et met ensuite la valeur du drapeau à 1.

Implémentation de TSL

fonction *test_and_set*(var lock: bool)

{registre$_i$ = lock ;
lock=true ;
return (registre$_i$) ;
}

Exemple de code

```
 {…….
while(test_and_set(lock)); /*attendre de façon active*/
Section critique;
lock=0;
……..
}
```

- Inconvénient ➔ attente active.

Les sémaphores

bouyakoub.f.m@gmail.com

Les sémaphores

- Introduit par *Dijkstra* en 1965 pour résoudre le problème de l'exclusion mutuelle;
- A une ressource critique sera associé un sémaphore;
- Règle le problème de l'attente active en introduisant un mécanisme de blocage des processus si la section critique est occupée, et de déblocage lors de la libération de la section critique.

Définition

- Un sémaphore S est une variable entière à laquelle est associée une file d'attente F(S).
- La variable peut prendre des valeurs positives, nulles ou négatives.
- La valeur initiale de la variable est toujours supérieure ou égale à 0.
- La politique de gestion de la file d'attente est généralement FIFO.
- On peut accéder au sémaphore au moyen des trois primitives:
  - I(S, x);
  - P(S);
  - V(S);

Primitives de manipulation des sémaphores

I(S, x)

```
{S=x;
F(S)=Ø;
}
```

La valeur de l'initialisation indique le nombre d'accès simultanés autorisés à la SC

P(S)

```
{ S--;
  Si (S < 0) alors bloquer le processus et le
                  mettre en queue de file F(S);
}
```

P(S) correspond à une tentative de franchissement. S'il n'y a pas de jeton pour la SC alors attendre, sinon prendre un jeton et entrer dans la SC.

V(S)

```
{ S++;
  Si (S <= 0) alors débloquer le processus
                  en tête de file F(S);
}
```

V(S) permet de rendre le jeton à la sortie de la SC et voir s'il y a des processus en attente.
Il est possible de déposer des jetons à l'avance.

Rôle de la variable du sémaphore

- L'initialisation de la variable dépend du nombre de processus pouvant accéder en **même temps** à une même "section critique";
- La valeur de la variable donne à chaque fois le nombre de ressources libres;
- Lorsque la valeur de la variable est négative, sa valeur absolue donne le nombre de processus dans la file d'attente.

Différents types de sémaphores

- Le type du sémaphore dépend de la valeur d'initialisation de son compteur:
  - ***Sémaphore binaire***;
  - ***Sémaphore de comptage***;
  - ***Sémaphore privé***.

Sémaphore binaire

- Un ***sémaphore binaire*** est un sémaphore dont la valeur initiale est égale à 1 (utilisé pour l'exclusion mutuelle).

Exemple:

```
I(S,1);    // initialisation de la variable à 1
Processus Pi
{.....
P(S);      // Demande d'accès à la section critique
<Section critique>
V(S);      // Libération de la section critique
.....}
```

Sémaphore de comptage

- Un ***sémaphore de comptage*** est un sémaphore dont la valeur initiale est supérieure à 1 (utilisé pour accéder à une ressource à n points d'accès).

Exemple:

```
I(S,n);   // initialisation de la variable à n
Processus Pi
{.....
P(S);      // Demande d'accès à la section critique
<Section critique>  // les n premiers processus accèdent à la SC
V(S);      // Libération de la section critique
.....}
```

Sémaphore privé

- Un ***sémaphore privé*** est un sémaphore dont la valeur initiale est égale à 0.
  - On dit que S est privé à un processus s'il est le seul à exécuter P(S), les autres ne peuvent exécuter que le V(S).

Exemple:

P1 et P2 qui coopèrent pour réaliser une tâche composée de deux parties: P1 réalise la première partie et P2 la deuxième.

I(S,0);

Processus P1

{…….
<Réaliser partie 1>
V(S);
…….}

Processus P2

{…….
P(S);
<Réaliser partie 2>
…….}

Exemple d'utilisation de sémaphores

Ecrire un programme qui permet de réguler l'entrée et la sortie d'un parking possédant N places. L'entrée et la sortie du parking ne laissent passer qu'un véhicule à la fois.

Code associé

```
I(NbPlace,N);
I(Entrée,1);
I(Sortie=1);
P(NbPlace);  // place disponible ???
P(Entrée);
<entrée parking>;
V(Entrée);
<Se garer>;
P(Sortie);
<Sortie parking>;
V(Sortie);
V(NbPlace);  // libérer la place
```

Inconvénients des sémaphores (1/2)

- Risque d'interblocage (*deadlock*) lors de l'utilisation de deux SC imbriquées.
  - Exemple: Init (S1,1); Init(S2,1);

Processus P1		Processus P2	
{…….		{…….	
P(S1);	➔ T1	P(S2);	➔ T'1
P(S2);	➔ T2	P(S1);	➔ T'2
…….		…….	
V(S2);		V(S1);	
V(S1);		V(S2);	
…….}		…….}	

Scénario possible: T1, T'1, T2, T'2 ➔ Interblocage

Inconvénients des sémaphores (2/2)

- Attente indéfinie en SC: la validité de la solution repose sur l'hypothèse que "tout processus sort de la SC au bout d'un temps fini".

Exemples classiques

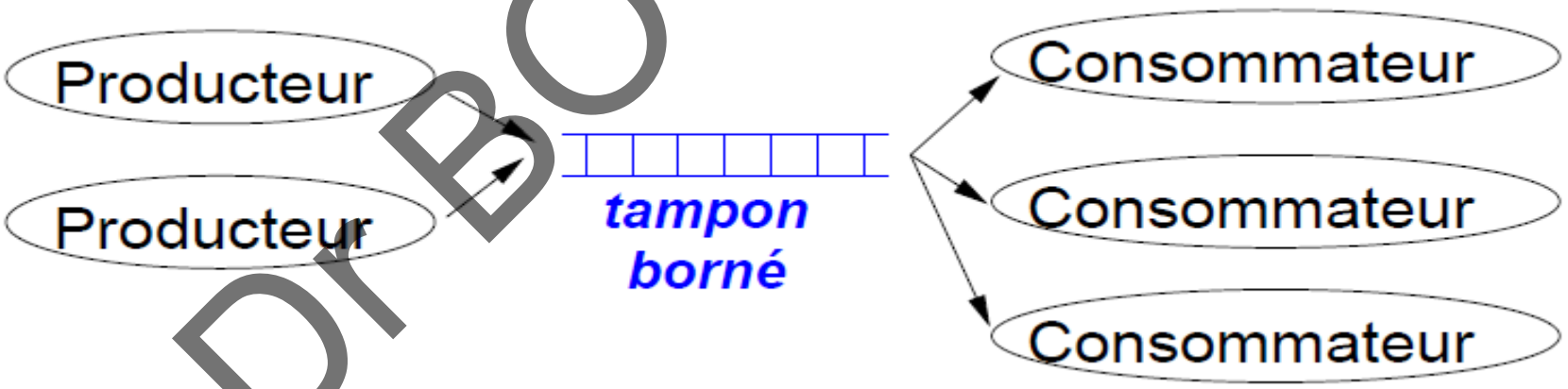
1. Producteurs/consommateurs;
2. Lecteurs/ rédacteurs;
3. Le dîner des philosophes.

Exemple du producteurs/consommateurs

- On dispose de deux catégories de processus (des producteurs et des consommateurs) qui se partagent une zone mémoire.
- Les producteurs remplissent la mémoire partagée avec des éléments.
- Les consommateurs récupèrent le contenu de la mémoire partagée.

Contraintes du producteurs/ consommateurs

- Un producteur doit se bloquer lorsque la mémoire partagée est pleine.
- Un consommateur doit se bloquer lorsque la mémoire partagée est vide.



Solution du producteurs/ consommateurs

- Utilisation de deux sémaphores, implémentant les deux critères de blocage:
  - l'un représente le nombre de cases libres;
  - l'autre représente le nombre de cases occupées;
- Utiliser un mutex pour l'accès à la mémoire partagée.

Prototype de solution

Initialisation des sémaphores

I(mutex, 1); I(vide, N); I(plein, 0);

Processus Producteur

```
{
// est ce qu'il existe des cases vides?
P(vide);
//Accès à la section critique
P(mutex);
//produire
tampon[prod] = 1;
prod=(prod + 1) mod N;
// libérer la SC
V(mutex);
// incrémenter le nombre de cases occupées
V(plein);
}
```

Processus Consommateur

```
{
// est ce qu'il existe des cases occupées?
P(plein);
//Accès à la section critique
P(mutex);
//consommer
x  = tampon[cons];
cons  =  (cons + 1) mod N;
// libérer la SC
V(mutex);
// incrémenter le nombre de cases vides
V(vide);
}
```

Exemples classiques

1. Producteurs/consommateurs;
2. Lecteurs/ rédacteurs;
3. Le dîner des philosophes.

Exemple des lecteurs/rédacteurs

- Deux catégories de processus (des lecteurs et des rédacteurs), qui se partagent une zone mémoire (fichier, un enregistrement dans un fichier ou une base de données).

- Les lecteurs font des accès en lecture seule à cette zone.

- Les rédacteurs modifient le contenu de cette zone.

Contraintes

- Lorsqu'un rédacteur accède à la mémoire partagée, aucun autre processus (qu'il soit lecteur ou rédacteur) ne doit y avoir accès ➔ problème d'exclusion mutuelle classique.
- En revanche, les lecteurs peuvent être plusieurs à utiliser la zone en même temps.

- Solution: On protège la mémoire partagée par une exclusion mutuelle, mais:
  - les lecteurs n'ont besoin de cette exclusion mutuelle **que si** aucun autre rédacteur n'utilise la mémoire ;
  - pour le vérifier, ils utilisent un compteur (nombre de rédacteurs en train d'utiliser la zone), lui aussi protégé par une exclusion mutuelle.

Variantes du problèmes des lecteurs/rédacteurs

- ***Priorité aux lecteurs***: S'il existe des lecteurs sur le fichier, toute nouvelle demande de lecture est acceptée ➔ **risque** : le rédacteur peut ne jamais accéder au fichier (famine).
- ***Priorité aux lecteurs, sans famine des rédacteurs***: S'il existe des lecteurs sur le fichier, toute nouvelle demande de lecture est acceptée sauf s'il y a un rédacteur en attente.
- ***Priorité aux rédacteurs***: Un lecteur ne peut lire que si aucun rédacteur n'est présent ou en attente ➔ risque de famine.
- ***FIFO***: Les demandes d'accès à l'objet sont servies dans l'ordre d'arrivée. S'il y a plusieurs lecteurs consécutifs, ils sont servis ensemble.

Exemples classiques

1. Producteurs/consommateurs;
2. Lecteurs/ rédacteurs;
3. Le dîner des philosophes.

Exemple du dîner des philosophes

- Cinq philosophes passent leur vie à penser/parler et manger autour d'une table.
- Pour manger, ils ont besoin de deux fourchettes, mais il n'y a que cinq fourchettes.

Contraintes

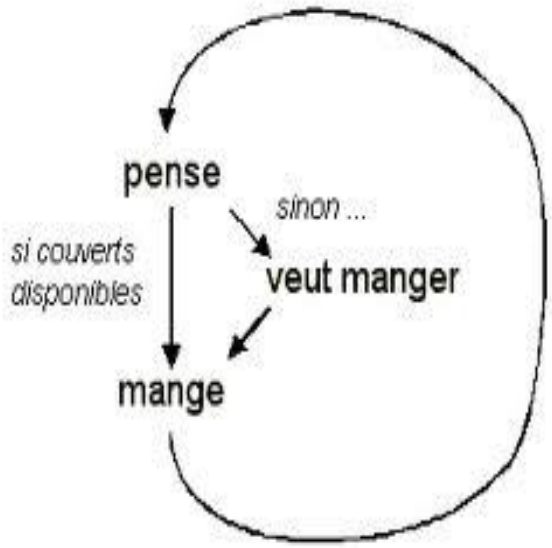
- Un philosophe, lorsqu'il a faim, prend la fourchette de gauche, puis celle de droite, mange, puis les repose ➔Risque d'interblocage.

Solution:

3 états pour un philosophe: "je pense", "j'ai faim", "je mange".

Lorsqu'il a faim, un philosophe ne peut manger que si ses deux voisins ne mangent pas, sinon il attend.

Lorsqu'il termine de manger, le philosophe réveille ses voisins et se remet à penser.



Les évènements mémorisés

Les évènements mémorisés

- Les mécanismes de synchronisation doivent permettre à un processus:
  - De bloquer un autre ou se bloquer lui-même, en attendant l'arrivée d'un signal d'un autre processus;
  - D'activer un processus.
- Deux cas se présentent:
  - Soit le signal n'est pas mémorisé et par conséquent il est perdu si le processus ne l'attend pas;
  - Soit le signal est mémorisé et par conséquent le processus ne se bloquera pas lors de sa prochaine opération.

Définition d'un évènement mémorisé

- Un évènement mémorisé "e" est une variable qui peut prendre deux valeurs: "**arrivé**" et "**non arrivé**".
- Deux opérations sont possibles sur les évènements:
  - e← <valeur> // arrivé ou non arrivé
  - Attendre(e); //**si** (e=="non arrivé") alors bloquer le processus P **fsi**
- Lorsqu'un EM reçoit la valeur "**arrivé**", tous les processus en attente de cet évènement passent à l'état prêt.

Exemple 1

Processus P1

```
{
.......
Ecrire(a);
e ← arrivé;
.......
}
```

Processus P2

```
{
.......
Attendre(e);
Lire(a);
.......
}
```

Exemple 2

N: entier initialisé à 0;
NbProcessus est le nombre de processus;

Processus Pi

```
{
……..
N++;
Si (N < NbProcessus)
  alors Attendre(e);
  sinon e← arrivé;
fsi
……..
}
```

Questions ?