Systèmes d'exploitation

Chapitre 3: Ordonnancement et Synchronisation des Processus

I. Ordonnancement

PROBLEMATIQUE :

Un ensemble de processus p_1, p_2,…, p_n en état prêt et qui attendent le processeur et on ne dispose que d'un seul processeur.

Solution :

Ordre d'exécution des processus (Ordonnancement des processus).

ROLE D`UN ORDONNENCEUR

Il définit l'ordre dans lequel les processus s'exécutent par le processeur et la durée pendant laquelle ils s'exécutent. Un bon ordonnanceur doit être capable d'assurer :

a. Equité :

Chaque processus doit avoir du temps du processeur.

b. Efficacité :

Le processeur doit être utilisé à 100%.

c. Temps de réponse (TR) :

Minimiser le temps de réponse pour les processus qui demandent à être exécutés .

d. Temps d'exécution (TE) :

Minimiser le temps d'exécution de plusieurs processus.

e. Débit :

Maximiser le nombre de processus traités.

TYPES D´ORDONNANCEURS

1-Ordonnancement sans réquisition :

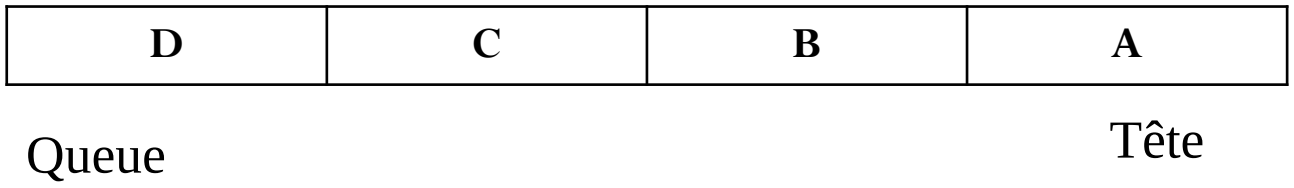
Un processus est exécuté jusqu'à la fin sans suspension.

1-1-L'ordonnanceur FIFO (First In First Out):

Les processus sont ordonnés suivant leur date d'arrivée au système.

Avantage : Algorithme facile à mettre en œuvre.

Exemple d'une File d'attente FIFO:



avec: T(A) <T(B) <T(C) <T(D) / T = Temps d´arrivé

Inconvénient : un processus avec un temps d'exécution (TE) minimum risque d'attendre longtemps avant d'être exécuté.

1-2-Ordonnanceur SJF (Short Job First):

- ✓Acronyme du plus court d'abord, la file d'attente doit être ordonnée selon le TE.
- ✓Le premier processus exécuté est le premier processus qui a le plus court TE.

Inconvénient : Les processus ayant un TE très grand risquent de ne jamais être exécutés.

Exemple:

1)

8 min	4 min	4 min	4 min
A	B	C	D

Tête

2)

4 min	4 min	4 min	8 min
B	C	D	A

Tête Queue

Queue

Avec l'ordre 1) :

Le temps écoulé après l'exécution de A est de 8min, de 12min pour B, de 16min pour C et de 20min pour D. Le temps moyen d'exécution (TME) = 14 min.

Avec l'ordre 2) :

Les temps écoulés sont 4, 8, 12 et 20min et la moyenne est 11min.

Il est plus optimal.

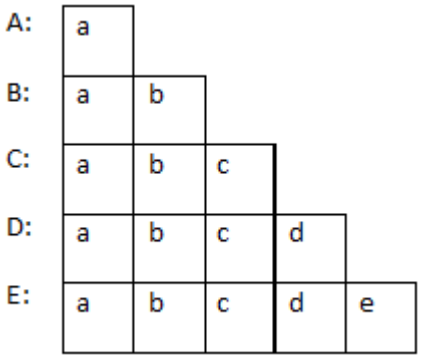
Temps moyen d'attente des processus (TMA):

Soient les processus A, B, C, D et E.

File d'exécution des processus (P) avec temps d'exécution (TE):

TE	e min	d min	c min	b min	a min
P	E	D	C	B	A

Queue Tête



$$TME = \frac{T(A)+T(B)+T(C)+T(D)+T(E)}{5}$$
$$= \frac{a+(a+b)+(a+b+c)+(a+b+c+d)+a+b+c+e)}{5}$$
$$= \frac{5a+4b+3c+2d+e}{5}$$

$$TMA = \frac{T(A)+\big(T(A)+T(B)\big)+\big(T(A)+T(B)+T(C)\big)+(T(A)+T(B)+T(C)+T(D))}{5}$$
$$= \frac{a+(a+b)+(a+b+c)+(a+b+c+d)}{5}$$
$$= \frac{4a+3b+2c+d}{5}$$

Exemple:

Soient :

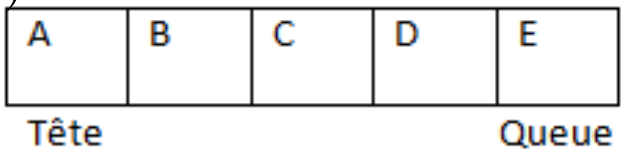
P	A	B	C	D	E
TA	0	0	1	1	2
TE	7	5	8	4	3

Question :

donner les files d'exécution pour les algorithmes FIFO et SJF et calculer les temps moyen d'exécution.

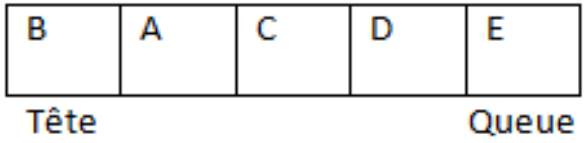
Avec FIFO

1)



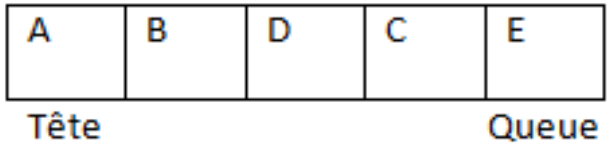
$$TM=\frac{35+20+24+8+3}{5}=18$$

2)



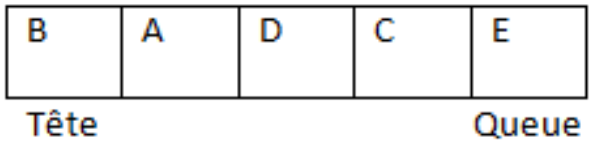
$$TM=\frac{25+28+24+8+3}{5}=17.6$$

3)



$$TM=\frac{35+20+12+16+3}{5}=17.2$$

4)



$$TM=\frac{25+28+12+16+3}{5}=16.8$$

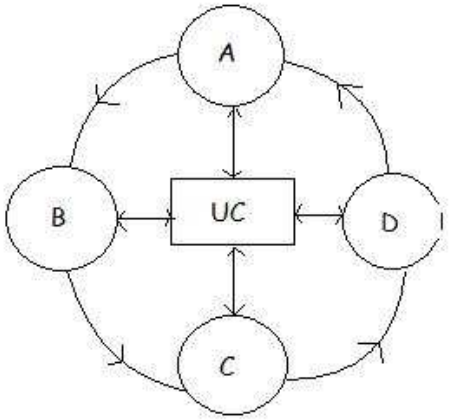
Avec SJF

2. Ordonnancement avec requisition:

Définition:

- Le SE reprend la main à chaque signal d'horloge et décide si le processus en cours d'exécution a consommé son quota de temps (unité de temps) et alloue le processeur à un autre processus.
- C'est une stratégie à temps partagé.
- L'exécution d'un processus est interrompue au bout d'un certain temps.

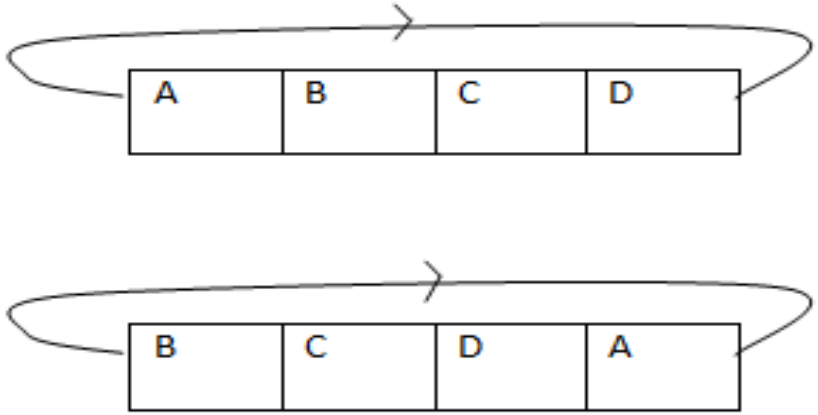
Ordonnanceur circulaire (Round-Robin) (Tourniquet) :



A,B,C,D : processus
UC: Processeur
Q: quantum d'allocation

Principe: Un processus ne peut s'exécuter plus qu'un temps Q.

File d'attente circulaire :

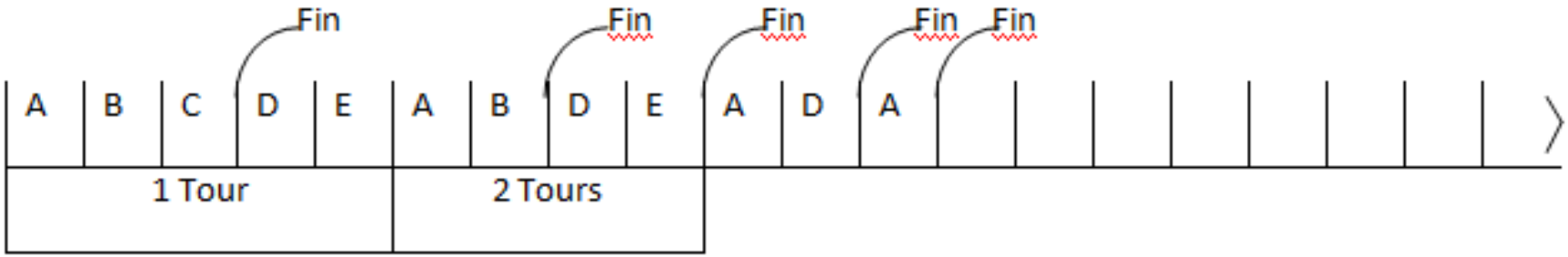


Exemple 1:

	A	B	C	D	E
TE	8	4	2	6	4

Q=2

File d'exécution :



$$TME = \frac{T(A)+T(B)+T(C)+T(D)+T(E)}{5}$$

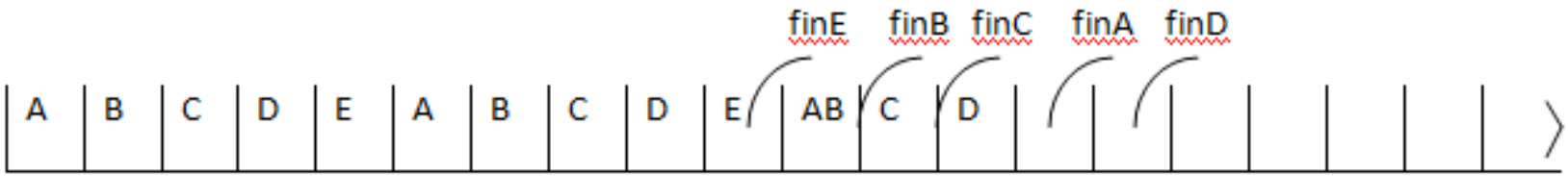
$$= \frac{24+14+6+22+18}{5} = 16.8$$

Exemple 2:

	A	B	C	D	E
TE	7	5	6	8	3

Q=2

File d'exécution :



$$TME = \frac{T(A)+T(B)+T(C)+T(D)+T(E)}{5}$$

$$= \frac{27+22+24+29+19}{5} = 24.2$$

3. Ordonnancement avec Priorité:

3.1-Algorithmes sans réquisition:

Définition:

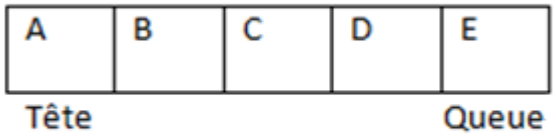
L'algorithme priorité sans réquisition permet aux processus prioritaire de s'exécuter en premier.

Exemple :

	A	B	C	D	E
Priorité	1	2	1	2	3

Remarque: les processus prioritaire sont ceux dont la priorité est minimale.

File d'attente :



$$TME=\frac{5A+4C+3B+2D+E}{5}$$

3.2 - Algorithmes avec réquisition:

Définition:

L'algorithme priorité avec réquisition permet aux processus prioritaire de s'exécuter en premier comme suit :

1) Il utilise plusieurs fils d'attentes (en nombre de priorités existantes).
2) Il stocke les processus de priorité i dans la file d'attente i.
3) Il exécute le processus j de priorité i durant son quantum et le place dans la file d'attente i+1.

Exemple 1: Soient:

	P_1	P_2	P_3	P_4	P_5
TE	120	350	500	50	200
Priorité	2	3	3	1	1

Q_1=100 Q_2=150 Q_3=200

Question: donner la file d'exécution et déterminer le TME.

Exemple 2: Soient:

	A	B	C	D	E
TE	9	6	3	5	X

Question:

Quel est le meilleur ordre d'exécution pour X par SJF?

Exemple 3:

Simuler l'exécution avec un ordonnanceur SJF des cinq jobs notés de A à E respectivement ayant des TEs 2, 4, 1, 1, 1 unité T respectivement.

Ces travaux sont soumis respectivement aux instants 0, 0, 3, 3, 3.

Calculer le TME dans ce contexte puis dans un contexte où les travaux sont soumis au même temps.

Notion d'interruption

Se fait par un signal envoyé à un processeur pour indiquer qu'un événement asynchrone (qui peut survenir à n'importe quel moment) s'est produit.

La séquence courante des instructions est temporairement suspendue et une séquence appropriée à l'interruption commence à sa place.

Plusieurs catégories d'interruption:

- Appel au superviseur par un processus,
- (SVC: SuperVisor Call),
- Déroutement (violation de protection mémoire, instruction inconue, division par zéro, …),
- Interruption due à l'opérateur,
- Synchronisation à l'aide de l'horloge,
- Dispositifs d'E/S,

Interaction entre les processus

- Communications entre processus pour échanger des informations génèrent généralement un Accès concurrent à des ressources partageables, exemple d'accès concurrent: cas du SPOOL
- Une solution au problème d'accès concurrent à une section critique (partie du programme où il risque d'avoir conflit d'accès) est l'exclusion mutuelle.

Quelques méthodes pour réaliser l'exclusion mutuelle:

1. Masquage des interruptions
2. Variables de verrouillage
3. L'Alternance

4. L'algorithme de PETERSON
5. L'instruction TSL (Test and Set Lock)
6. Les primitives SLEEP & WAKEUP
7. Les sémaphores
8. Les moniteurs
9. L'échange de messages

Masquage des interruptions:

✓Avant d'entrer dans une section critique, le processus **masque les interruptions.**

✓Il les restaure à la fin de la section critique. Il ne peut être alors suspendu durant l'exécution de la section critique.

✓ Cependant cette solution est dangereuse, car si le processus, pour une raison ou pour une autre, ne restaure pas les interruptions à la sortie de la section critique, ce serait la fin du système.

✓La solution n'assure pas l'exclusion mutuelle, si le système n'est pas monoprocesseur car le masquage des interruptions concernera uniquement le processeur qui a demandé l'interdiction.

✓Les autres processus exécutés par un autre processeur pourront donc accéder aux objets partagés.

✓En revanche, cette technique est parfois utilisée par le système d'exploitation pour mettre à jour des variables ou des listes partagées par ses processus, par exemple la liste des processus prêts.

Variables de verrouillage (Verrou):

Verrou variable unique partagée entre processus.

Algorithme:

Verrou initialisé à 0;

```
Si verrou = 0 alors
        mettre verrou à 1
        section critique
        remettre verrou à 0
Sinon (cas où verrou = 1) attendre que verrou passe à 0
```

Situation qui mène à un problème d'interblocage !!! :

Supposons que le processus A a lu le verrou et trouve 0 avant de mettre verrou à 1, le processus A est interrompu ,

Le processus B qui a la main trouve verrou à 0
B met verrou à 1
Entre dans la section critique !!!
Puis B est interrompu

Le processus A prend la main
A trouve toujours verrou à 0
il met verrou à 1
A rentre en section critique !!!

Donc Nous avons deux processus en section critique !!!!

Exclusion mutuelle avec un drapeau (verrou) pour chaque processus:

Processus A	Processus B
Faire toujours	Faire toujours
(1) partie neutre	(1') partie neutre
(2) occup1:= true	(2') occup2:= true
(3) Tant que occup2	(3') Tant que occup1
attendre	attendre
section critique	section critique
occup1:=false	occup2:= false

Les deux processus A et B se bloquent mutuellement en (3) et (3') (deadlock)

L'Alternance

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

Soient deux processus A et B. La variable tour est initialisée à 0.

Processus A	Processus B
while tour <> 0 attendre	while tour <> 1 attendre
section critique	section critique
tour = 1	tour = 0
section non critique	section non critique

Problème :

Supposons que le processus A quitte sa section critique ➔ tour=1

Le processus B entre et quitte aussi sa section critique ➔ tour = 0

Le processus A peut alors entrer en section critique
il y entre puis il sort ➔ tour =1
le processus A est dans sa section non critique

Le processus B est encore dans sa section non critique
il y reste pendant un moment

Si le processus A veut entrer en section critique
IL NE PEUT PAS car tour est toujours = 1

Donc le processus B n'est pas en section critique mais IL BLOQUE A
à entrer dans sa section critique !!!

Donc :

✓ On peut vérifier assez facilement que deux processus ne peuvent entrer en section critique en même temps,

✓ Toutefois le problème n'est pas vraiment résolu car il est possible qu'un des deux processus ait plus souvent besoin d'entrer en section critique que l'autre ; l'algorithme lui fera attendre son tour bien que la section critique ne soit pas utilisée.

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

L´Algorithme de PETERSON

✓La solution de Peterson se base sur l'utilisation de deux fonctions :
entrer_region(); quitter_region();

✓Chaque processus doit, avant d'entrer dans sa section critique appeler la fonction entrer_region() en lui fournissant en paramètre son numéro de processus.

```
section non critique;
entrer_region();
section  critique;
quitter_region();
section non critique;
```

✓Cet appel le fera attendre si nécessaire 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, comme le montre le code peterson.c.

```
#define FALSE 0
#define TRUE  1
#define N     2      // Nb. processus

int  tour;           // Le tour du processus
int interesse[N];    // 0 initiale

void entrer_region(int processus)    // processus 0 ou 1
{
    int autre ;      // Autre processus
    autre = 1 - processus ;
    interesse[processus] = TRUE ;
    tour = processus ;  // Positioner le drapeau
                        // On est  interessé
    while (tour == processus && interesse[autre] ==TRUE) ;
}

void quitter_region (int processus)
{
    // Processus quite region critique
    interesse[processus] = FALSE;
}
```

L´Instruction (Test and Set Lock)

on fait appel au hardware

instruction TSL:

change le contenu d'un mot mémoire dans un registre
met une valeur non nulle à l'adresse du mot

les opérations de lecture et d'écriture du mot sont INDIVISIBLES

drapeau variable partagée

entrer_region:

```
tsl  regestre,drapeau < == > registre <= drapeau
                          drapeau  < = 1
cmp   regitre,0
bne  entrer_region
ret
```

quitter_region:

```
mov drapeau,0
ret
```

Remarques

L'algorithme de PETERSON et l'instruction TSL ont l'inconvénient de l'attente active (test répété sur une variable pour détecter l'apparition d'une valeur.

Cette attente doit être évitée car elle consomme du temps CPU

Dans certains cas il y a risque de blocage. En effet:

Soient A et B deux processus avec priorité(A) > priorité(B)

A s'exécute et entre dans sa section critique

B passe à l'état prêt et il est élu (plus prioritaire)
(remarque: A est toujours dans sa section critique)

B veut entrer en section critique et il boucle sur
attente active

A ne peut pas continuer car B mobilise le CPU

Donc blocage de A et B

Primitives SLEEP & WEAKEUP

Au lieu de faire une attente active, les processus vont se bloquer au cas où ils ne peuvent pas entrer en section critique

SLEEP: appel système qui suspend le processus appelant jusqu'à ce qu'un autre processus vient le réveiller

WAKEUP (process): appel système qui réveille le processus passé en paramètre

Cas du PRODUCTEUR et du CONSOMMATEUR

```
#define N 100  /* taille du buffer */
int compteur = 0  /* nombre d'objets ds buffer */

producteur()
{while TRUE
  { produire_objet
   if (compteur==N) sleep()  /* buffer plein */
   mettre_objet
   compteur=compteur+1
   if (compteur==1)   /* compteur était à 0 */
     wakeup(consommateur) } }

consommateur()
{while TRUE
     { if (compteur==0) sleep()
   retirer_objet
   compteur=compteur-1
   if (compteur==N-1)   /* avant compteur=N */
     wakeup(producteur)
}}
```

Exécution des processus en parallèle ou (pseudo //)

Pour que des objets qui s'exécutent en « parallèle » et qui utilisent des objets partagées puissent coopérer correctement (sans conflit d'accès, ni blocage mutuel, ni attente indéfinie), il faut avoir les 4 conditions suivantes:

- 2 processus ne peuvent pas être en section critique en même temps,
- Aucun processus suspendu en dehors d'une section critique ne doit bloquer les autres processus à entrer en section critique,
- Aucun processus ne doit attendre trop longtemps à entrer en section critique,
- Aucune hypothèse n'est faite sur les vitesses des processus ni sur le nombre de processeurs

Les Sémaphores

✓ Solution proposée par Dijkstra pour compter le nombre d'appels en attente

✓ Un sémaphore S est 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 processeurs en attendant qu'une condition soit réalisée pour leur réveil.

✓Les processus bloqués sont placés dans la file d'attente

✓ Dijkstra proposa deux primitives appelées DOWN et UP

Les sémaphores permettent de réaliser des exclusions mutuelles de la façon suivante:

semaphore mutex = 1;		
processus P1: DOWN(mutex) section critique de P1; UP(mutex);		processus P2: DOWN(mutex) section critique de P2; UP(mutex);

DOWN et UP sont définies comme suit:

DOWN(S):

S ← S-1

si S < 0 alors bloquer le processus dans la file (S)

UP(S):

S ← S+1

si S ≤ 0 alors débloquer le processus de la file (S)

Un sémaphore ne peut être manipulé que par DOWN et UP

La valeur du compteur et l'état de la file sont inaccessibles

Exemple : Donner l´ordre d´exécution des 3 processus suivants

Val(S) = 1 et file(S) = □□□□

Processus 1	Processus 2	Processus 3
DOWN(S)	DOWN(S)	DOWN(S)
<SC>	<SC>	<SC>
UP(S)	UP(S)	UP(S)

Exemple :

Val(S) = 1 et file(S) = □□□□

Processus 1	Processus 2	Processus 3
(3) DOWN(S)	(1) DOWN(S)	(4) DOWN(S)
(6) <SC>	(2) <SC>	(8) <SC>
(7) UP(S)	(5) UP(S)	(9) UP(S)

Solution du producteur – consommateur avec les sémaphores

```
#define N 100   /* taille du buffer */
typedef int semaphore  /* définition de type */
semaphore mutex = 1  /* controle accès S.C. */
semaphore libre = N  /* Nbre places libres */
semaphore plein = 0  /* Nbre places occupées */
producteur ()
{
while TRUE
  {
  produire-objet  /* produire un objet */
  down(libre)  /* decrémenter nbre places libres */
  down(mutex)  /* entrer en S.C. */
  mettre-objet
  up(mutex)   /* sortir de la S.C. */
  up(plein)   /* incrémenter nbre places occupées */
   }
}
consommateur()

{
while TRUE
  {
  down(plein)  /* decrémenter nbre places occupées */
  down(mutex)  /* entrer en S.C. */
  retirer-objet
  up(mutex)   /* sortir de la S.C. */
  up(libre)   /* incrémenter nbre places libres */
   }
}
```

Les moniteurs

Un moniteur est une primitive de synchronisation

Un moniteur = ensemble de variables + procédures qui les utilisent

Les variables sont manipulées uniquement par les procédures du moniteur (non accessibles aux processus)

Les moniteurs font partie du langage de programmation

Dans un moniteur, le blocage et le réveil des processus se fait au moyen de conditions

Une condition est une variable qui ne peut être manipulé qu'à l'aide des opérations:

wait (C) : bloquer le processus et le placer en attente de C

signal (C) : réveiller un processus en attente de C

vide (C) : fonction booléenne qui est vraie si aucun processus n'attend C, fausse autrement

```
moniteur Producteur-Consommateur
condition plein, vide;
integer compteur;

procedure mettre;
begin
if compteur = N then wait(plein);
mettre-objet;
compteur:=compteur+1;
if compteur=1 then signal(vide);
end;

procedure retirer;
begin
if compteur = 0 then wait(vide);
retirer-objet;
compteur:=compteur-1;
if compteur=N-1 then signal(plein);
end;
compteur:=0;
endmoniteur;
```

```
procedure  producteur;
begin
while TRUE do
begin
   produite_objet;
   producteur_consommateur.mettre;
 end;
 end;

procedure consommateur;
begin
while TRUE do
begin
  producteur_consommateur.retirer;
 utiliser_objet;
end;
end;
```

L´Echange de messages

utilisé pour une communication inter-processus dans un système à plusieurs machines

2 primitives sont utilisées:

- send (destination, &message)

- receive (source, &message)

message d'acquittement

Modèle du producteur-consommateur avec N messages

```
#define N 100


producteur()
{ int objet;
message m;   /* buffer des messages */
while true
{
produire_objet (&objet)
receive(consommateur,&m)
faire_message(&m,objet)
send(consommateur,&m)
}
}

consommateur()
{
int objet,i;
message m;
for (i=0; i<N ; i++)
  send(producteur,&m);  /*envoyer m messages vides*/
while TRUE
{
receive(producteur,&m);  /*attendre un message*/
retirer_objet(&m,&objet); /*retirer l'objet du message*/
utiliser_objet(objet);  /* utiliser objet */
send(producteur,&m);  /*renvoyer une réponse vide*/
}
```

PROBLEME DES PHILOSOPHES

✓Il s'agit d'un problème théorique très intéressant proposé et résolu aussi par Dijkstra.

✓Cinq philosophes sont assis autour d'une table. Sur la table, il y a alternativement cinq plats de spaghettis et cinq fourchettes (Figure suivante).

✓Un philosophe passe son temps à manger et à penser.

✓Pour manger son plat de spaghettis, un philosophe a besoin de deux fourchettes qui sont de part et d'autre de son plat.

✓Il est important de signaler que si tous les philosophes prennent en même temps chacun une fourchette, aucun d'entre eux ne pourra prendre l'autre fourchette (situation d'interblocage).

✓Pour éviter cette situation, 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.

```
#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);
  }
  }
```

Risque de blocage si tous les philosophes prennent une fourchette en même temps

Avec Exclusion Mutuelle

```
#define N 5
typedef int semaphore /* définition de type */
semaphore mutex = 1 /* controle accès S.C. */
philosophe(i)
int i;
{
while TRUE
{
  penser();
  down(mutex);
  prendre_fourchette(i);
  prendre_fourchette((i+1)%N);
  manger();
  poser_fourchette(i);
  poser_fourchette((i+1)%N);
  up(mutex);
}
}
```

solution correcte mais non performante

Solution au problème des Philosophes

```
#define N  5  /* nbre de philosophes  */
#define 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;  /* type sémaphore (entier) */
int etat[N];    /* etat de chaque philo init PENSE*/
semaphore mutex = 1;  /* exclusion mutuelle */
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);
etat[i]=FAIM;
test(i);
up(mutex);
down(s[i]);
```

```
poser_fourchettes(i)
int i;
{
down(mutex);
etat[i]=PENSE;
test(GAUCHE);
test(DROITE);
}

test(i)
int i;
{
if (etat[i]==FAIM)&&(etat[GAUCHE]!=MANGE)
   &&(etat[DROITE!=MANGE)
   { etat[i]=MANGE;
     up(s[i]);
   }
}
```

Problème des Lecteurs et Rédacteurs

```
typedef int semaphore;
semaphore mutex=1;
semaphore base=1;
int nbrelire=0;

lecteur()
{
while TRUE
{
  down(mutex);
  nbrelire=nbrelire+1;
  if nbrelire==1)
down(base);
  up(mutex);
  lire_base();
  down(mutex);
  nbrelire=nbrelire-1;
  if (nbrelire==0) up(base);
  up(mutex);
  utiliser_données_lues;
}
}
```

```
redacteur()
{
while TRUE
{
  creer_donnees();
  down(base);
  ecrire_donees();
  up(base);
}
}
```

Les Interblocages

Les 4 conditions de Coffman qu'il faut réunir pour qu'il puisse y avoir interblocage sont:

1) Exclusion mutuelle: chaque ressource est soit attribuée à un seul processus, soit disponible,
2) La détention et l'attente: les processus qui détiennent des ressources peuvent en demander d'autres ressources,
3) Pas de réquisition: les ressources obtenues par processus ne peuvent lui être retirées, elles ne sont disponibles que s'il les libère,
4) L'attente circulaire: il doit y avoir une chaine circulaire de deux ou plusieurs processus, chacun d'entre eux attendant une ressource détendue par le processus suivant dans la chaine.

Traitement des interblocages

- Pas de politique pour éviter ou traiter les interblocages,
- Technique de la détection et de la reprise,
- Prévenir les interblocages,
- Éviter les interblocages (algorithme du banquier),

Algorithme du banquier pour une ressource unique

On appelle état du système concernant l'allocation des ressources, la liste:

- Des clients (processus),
- Des montants empruntés (nombre de ressources déjà allouées),
- Des crédits maximums auxquels les clients ont droit (nombre de ressources non encore allouées et dont les processus auront besoin)

Un état des dit sûr, s'il existe une suite d'états permettant aux clients de disposer de leur crédit maximum.

exemple

ETAT 0

processus	ress. utilisées	ress. max
P1	0	6
P2	0	5
P3	0	4
P4	0	7

Valeur de crédit maximum est: 6+5+4+7=22

Les processus n'auront pas besoin immédiatement de leurs ressources maximum

Supposons que nous ne disposons que de 10 instances de la ressource

Supposons que les processus p1, p2, P », p4 ont demandé respectivement les quantités de ressources 1, 1, 2, 4.

si on satisfait ces demandes on aura l'état suivant:

ETAT 1

processus	ress. utilisées	ress. max
P1	1	5
P2	1	4
P3	2	2
P4	4	3

Les ressources disponibles sont égales à:
ressources maximales – ressources utilisées = 10 - (1+1+2+4) = 2
l'état 1 est sûr car:

on ne peut faire attendre tous les processus sauf p3 auquel on alloue les deux unités restantes (p3 peut disposer de 4 ressources au maximum)

p3 se termine et libère les 4 unités dont il dispose,

on peut alors satisfaire p2 ou p4, et ainsi de suite
Supposons qu'on est à l'état 1 et que p2 demande une quantité de ressources égale à 1

si on satisfait cette demande est –ce qu'on reste dans un état sûr?

ETAT 2

processus	ress. utilisées	ress. max
P1	1	5
P2	2	2
P3	2	2
P4	4	3

les ressources disponibles sont égales à : 10 – (1+ 2 + 2+ 4) = 1

l'état 2 n'est pas sûr car aucun processus ne peut être satisfait s'il demande ces ressources maximales donc il y a **interblocage !**

Algorithme du banquier pour plusieurs ressources

ressources attribuées (ETAT A)

P	D	T	I	R
P1	3	0	1	1
P2	0	1	0	0
P3	1	1	1	0
P4	1	1	0	1
P5	0	0	0	0

P: Processus
D: Dérouleur
T: table
I: imprimante
R: perforateur

ressources exist.

E = 6 3 4 2

ressources affect.

A = 5 3 2 2

ressources demandées

P	D	T	I	R
P1	1	1	0	0
P2	0	1	1	2
P3	3	1	0	0
P4	0	0	1	0
P5	2	1	1	0

ressources disp.

D = 1 0 2 0
(E-A)

L'ETAT A est sûr, en effet:

on peut satisfaire P4; alors D devient D + 1 1 0 1 = 2 1 2 1
on peut satisfaire P1; D = 2 1 2 1 + 3 0 1 1 = 5 1 3 2
on peut satisfaire P2; D = 5 1 3 2 + 0 1 0 0 = 5 2 3 2
on peut satisfaire P3; D = 5 2 3 2 + 1 1 1 0 = 6 3 4 2
on peut satisfaire P5; D = 6 3 4 2 + 0 0 0 0 = 6 3 4 2

Algorithme qui détermine si un état est sûr

a) Trouver une rangée dans la matrice RD (ressource s demandées) plus faible que le vecteur D (s'il n'existe aucune rangée alors il y a risque d'interblocage)
b) Supposer que le processus de cette rangée s'est terminé et ajouter ses ressources à D à partir de la matrice des ressources allouées RA
c) Si tous les processus peuvent se terminer (satisfaire (a) et (b)) alors l'état est sûr sinon l'état n'est pas sûr.

Supposons que P2 demande une imprimante.

On va essayer de satisfaire la demande et voir si l'état résultat reste sûr.

P	D	T	I	R
P1	3	0	1	1
P2	0	1	1	0
P3	1	1	1	0
P4	1	1	0	1
P5	0	0	0	0

RA : ressources attribuées

P	D	T	I	R
P1	1	1	0	0
P2	0	1	0	2
P3	3	1	0	0
P4	0	0	1	0
P5	2	1	1	0

RD : ressources demandées

Le vecteur des ressources disponibles devient:

D = 1 0 1 0

P4 peut se terminer
D = 1 0 1 0 + 1 1 0 1 = 2 1 1 1
P1 peut se terminer
D = 2 1 1 1 + 3 0 1 1 = 5 1 2 2
P2 peut se terminer
D = 5 1 2 2 + 0 1 1 0 = 5 2 3 2
P3 peut se terminer
D = 5 2 3 2 + 1 1 1 0 = 6 3 4 2
P5 peut se terminer
D = D + 0 0 0 0 = D

L'état résultant est un état sûr donc on peut satisfaire la demande.

Supposons que maintenant P5 demande une imprimante

D devient 1 0 1 0 – 0 0 1 0 = 1 0 0 0
et les matrices RA et RD seront comme suit:

RA

P	D	T	I	R
P1	3	0	1	1
P2	0	1	1	0
P3	1	1	1	0
P4	1	1	0	1
P5	0	0	1	0

RD

P	D	T	I	R
P1	1	1	0	0
P2	0	1	0	2
P3	3	1	0	0
P4	0	0	1	0
P5	2	1	0	0

aucune rangée de RD ne peut être satisfaite avec le vecteur D = 1 0 0 0
Donc il y a risque d'interblocage.