Communication entre tâches

Par partage de mémoire

Problématique de la communication par partage de mémoire

- Plusieurs tâches en concurrence
  - Uni processeur
    - M tâches pré-emptives sont ordonnancées par le système
  - Multi processeur
    - N processeurs supportant P tâches s'exécutant en parallèle
- Caractéristiques de la programmation concurrente
  - Les tâches peuvent avoir des performances différentes selon le choix de l'OS ou la prise en compte des E/S
  - L'exécution d'un flot est prédictible (séquence) ; l'exécution de plusieurs flots parallèles est non prédictible.
  - Les tâches s'exécutent indépendamment les unes des autres.

- Problèmes des exécutions concurrentes
  - Accès à des mémoires partagées
    - L'ordre des accès peut influencer le résultat final d'où
  - Nécessiter de coopérer et de se synchroniser
    - Accès exclusif à une ressource partagée pour maintenir la cohérence des données.

- Exemple 1
  - *Soit x une variable partagée*

  Tâche1 *Tâche2* *Tâche3*

  a) x=1 *b) x=2* *c) y=x*

  La valeur de y est imprévisible.

  Les séquences *c) a) b) => y = 0*

  b) a) c) => y = 1

  a) b) c) => y= 2

  sont possibles

- Exemple 2

```
#include <stdio.h>
#include <pthread.h>
pthread_t id1, id2 ;

int main () {
    int compte_client = 0;

    pthread_create(&id1, NULL, (void *(*)() f, NULL) ;
    pthread_create(&id2, NULL, (void *(*)() g, NULL) ;

    pthread_join (id1, NULL) ;
    pthread_join (id2, NULL) ;

}
```

• Exemple 3

/* le code des tâches */

```
void f(void) {
    int x = 0;        /*  variable locale */
    int j;
    for (j=0 ; j < 16 ; j++) {
          x = compte_client;          /*P1
          x = x +1 ;                  /*P2
          compte_client = x ;         /*P3
          }
}
```

```
void g(void) {
    int y = 0;        /*  variable locale */
    int j;
    for (j=0 ; j < 16 ; j++) {
          y = compte_client;          /*Q1
          y = y +1 ;                  /*Q2
          compte_client = y ;         /*Q3
          }
}
```

P1 Q1 Q2 P2 Q3 P3 Résultat = ?

Q1 Q2 Q3 P1 P2 P3 Résultat = ?

P1 P2 P3 Q1 Q2 Q3 Résultat = ?

- Exemple 4
  - *Soit x une variable partagée*

Tâche1
while (true) {
a) x=1 ;
}

Tâche 2
while (true) {
b) x = 2 ;
}

Tâche 3
while (true) {
c) if (x == 2) exit(0) ;
}

La séquence b) a) c) b) a) c) b) a) c) => la tâche 3 est toujours bloquée

- Exemple 5
  - *Soit y et z variables partagées ; y = z= 0;*

Tâche1
x = y+ z ;

Tâche2
y = 1 ;
z = 2 ;

x peut valoir 0, 1, 2 ou 3.

Exécution simultanée d'instructions machine

Soit diff variable commune diff = 0 ;

```
Tâche 1 {                          Tâche 2 {
int mydiff ; /* locale */         int mydiff ; /*locale */
mydiff = 1 ;                       mydiff = 1 ;
diff = mydiff + diff ;             diff = mydiff + diff ;
}                                  }
```

	Tâche 1 sur proc1	Tâche 2 sur proc2
1	→ R1 <- diff	→
2	→	→ R0 <- diff
3	→ R1 <- R1 + mydiff /*de P1 */	→
4	→	→ R0 <- R0 + mydiff /* de P2*/
5	→diff <- R1 /* diff = 1 */	→
6		→ diff <- R0 /* diff = 1 */

Temps

La valeur attendue de diff = 2

Quelle est la valeur produite par cette exécution ?

Exclusion mutuelle et section critique

- Exclusion mutuelle
  - Condition de fonctionnement garantissant à un processus l'accès exclusif à une ressource critique pendant une suite d'opérations avec cette ressource
- Section critique
  - Séquence d'actions d'une tâche pendant laquelle cette tâche est la seule à utiliser une ressource critique
  - Un segment de code qui accède à des variables partagées (ou des ressources partagées) devant être exécuté comme une action atomique est une section critique.

```
While (true) {
    entrée-en-section-critique
    section_critique // accès à des ressources partagées
    sortie-de-section-critique
    section_non_critique
}
```

Les sections entrée et sortie doivent satisfaire les conditions suivantes :

- Exclusion mutuelle : Quand une tâche s'exécute dans sa section critique, aucune autre tâche ne peut alors s'y trouver
- Progression : Si des tâches cherchent à entrer et qu'aucune autre tâche n'est en section critique, alors seules les tâches étant dans des opérations d'entrée ou sortie participent à l'élection, et la décision ne peut être retardée indéfiniment
- Attentes limitées : toute tâche sort de sa section critique au bout d'un temps fini.

- Corollaire :
  - Aucune hypothèse sur les synchronisation d'horloge des processeurs.
  - Aucune tâche suspendue en dehors d'une section critique ne doit bloquer les autres tâches.
  - Les tâches doivent entrer en SC le plus rapidement possible.

Implémentation

1) Masquage des interruptions
   - Dangereux, programmation système très ciblée
2) Algorithme : Variables verrou ou de synchronisation
   - Sujet à caution suivant la machine (hard+compilo)
3) Solution générale
   - Ressource matérielle
     - Instructions atomiques (test and set)
     - Barrières de synchronisations
   - Ressource logicielle
     - Sémaphore (primitive)
     - Moniteur (fonctions)

Solution algorithmique : les variables de verrouillage

- Un verrou, variable partagée, à une valeur initiale = 0 ;

(verrou = = 0) signifie aucun processus en SC, verrou = = 1 signifie un processus en SC.

- Un processeur teste le verrou ;
  - Si verrou == 0 alors verrou =1 et entrée en SC
  - Si verrou == 1 alors attente que verrou soit à 0.

Variable de synchronisation

- Soit sync une variable commune initialisée à 0
  - Si sync = 0 alors la tâche A entre en section critique sinon elle attend
  - Si sync =1 alors la tâche B entre en section critique sinon elle attend

```
while (true) {
        while (sync != 0) ;  /* attente que B me délivre */
        section_critique() ;
        sync = 1 :
        section_noncritique ;
}
```

```
while (true) {
        while (sync != 1) ;  /* attente que A me délivre
        section_critique() ;
        sync = 0 :
        section_noncritique ;
}
```


Solution ?

La tâche A (resp B) est bien bloquée par la tâche B (resp A) qui exécute une section non_critique car sync vaut 0 ou 1 mais pas les deux à la fois.

t

	Tâche A	**Tâche B**
	while (true) {	
t0	*while (sync != 0) ;*	*while (true) {*
t1	*section_critique() ;*	*t1..ti+1 while (sync != 1) ;*
ti	*sync = 1 :*	*tj section_critique() ;*
ti+1..tm	*section_noncritique ;*	*tj+1 sync = 0 :*
	}	*tj+2 section_noncritique ;*
		}

Non conforme à l'objectif de l'exclusion mutuelle : la tâche A peut être bloquée par une tâche B en train d'exécuter des instructions dans la section non – critique (A plus rapide que B).

Possibilité d'attentes illimitées si la tâche B, bloquée a été mise « en attente »

- ✓ t0 La tâche A lit sync et trouve 0 => entre en SC
- ✓ t1 la tâche B lit sync et trouve 0 => attente
- ✓ ti+1 la tâche A quitte sa SC et sync = 1
- ✓ ti+1 la tâche B est débloquée et
- ✓ tj+1 elle sort de sa SC => sync = 0

Deuxième itération :

- ✓ t0 à ti la tâche A entre dans sa SC, sort, et sync = 1
- ✓ la tâche A est en attente

Algorithme de Peterson

```
boolean flag0 = flag1 = false ;
int sync ;
```

Tâche A

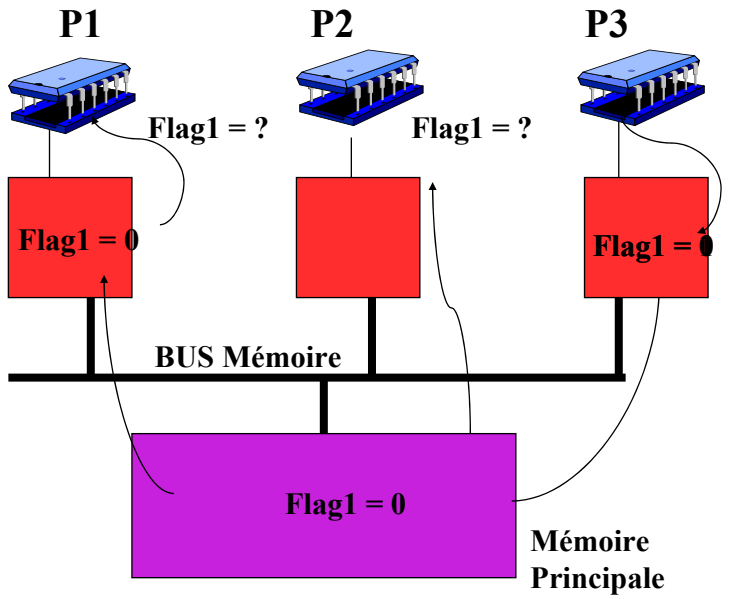
```
while (true) {
        flag0 = true ;
        sync = 1 ;
        while (flag1 and sync ==1) ;  /* déblocage si flag1 = false ou sync = 0*/
        section_critique() ;
        flag0 = false ;
        section_noncritique ;
}
```

Tâche B

```
  while (true) {
        flag1 = true ;
        sync = 0 ;
        while (flag0 and sync == 0) ;  /* déblocage si flag0 = false ou sync = 1*/
        section_critique() ;
        flag1 = false ;
        section_noncritique ;
  }
```

Solution ?

- Soit une architecture moderne à 3 processeurs, P1 exécutant la tâche 0 et P3 exécutant la tâche 1 chacun possédant son cache propre, P2 exécutant une tâche 2.
- Si flag0, flag1 et sync sont chargées en mémoire cache, les mises à jour effectuées par une tâche ne sont pas visibles par les autres tant que la mémoire centrale n'est pas mise à jour => solution matérielle (algorithme MESI).
- Si flag0, flag1 et sync sont compilées et utilisées par l'intermédiaire de registres de la machine, leurs mises à jour par une autre tâche peuvent ne pas être répercutées dans les registres.



```
flag0 = true ;
sync = 1 ;
while (flag1 and sync == 1) ;        ld    R2, flag0
                                     ld    R1,sync
                                     ld    R0, flag1

                              itere : cmp  R0,#0
                                      beq  itere
                                      cmp  R1,#1
                                      beq   itere
```

- Solution : faire en sorte que les variables partagées ne puissent être copiées dans des mémoires privées :

```
volatile boolean flag0 = flag1 = false ;
volatile int sync ;
```

- Usage intensif de l'UC dans des attentes actives qui sollicitent l'UC pour ne rien faire.

Solution : primitives matérielles pour gérer l'entrée et la sortie de SC

lock(verrou) : si verrou = 0 alors verrou = 1 et retour, sinon rien

unlock(verrou) verrou = 0

compilation en instructions machines conventionnelles :

```
lock :   ld    R0, verrou
         cmp   R0, #0
         bnz   lock
         st    verrou, #1
         rts

unlock : st    verrou , #0
         rts
```

Solution ?
Init : verrou = 0 (aucune tâche en section critique)

	P0		***P1***	
t0	*lock*	*ld R0, verrou*		
t1		*cmp R0, #0*		
t2			*lock*	*ld R0,verrou*
t3				*cmp R0,#0*
t4		*bnz lock*		
t5		*st verrou, #1*		
t6		*Entrée en SC*		
t7				*bnz lock*
t8				*st verrou, #1*
t9				*Entrée en SC*

- Solution : utilisation d'une synchronisation matérielle atomique
  - Exemple : l'instruction atomique *test_and_set (R0, verrou)*
    - Charge le contenu de verrou dans le registre R0
    - Si (R0 == 0) alors verrou <- 1 sinon rien
    - Fin de l'instruction

```
lock : test_and_set R0, verrou     instruction atomique pour tester
       cmp          R0, #0         et mettre à 1 le verrou.
       bnz          R0, lock
       rts                         retour pour entrer en SC

unlock :  st        verrou, #0
          rts
```

```
boolean lock = false ;
```

Tâche A

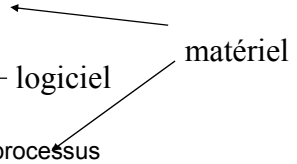
```
while (true) {
        function_lock(&lock) ;  /* rien tant que verrou pris*/
        section_critique() ;
        unlock(&lock) ;
        section_noncritique ;
}
```

Tâche B

```
    while (true) {
            function_lock(&lock) ;  /* rien tant que verrou pris*/
            section_critique() ;
            unlock(&lock) ;
            section_noncritique ;
    }
```


Réalisation de l'exclusion mutuelle

- L'acquisition du droit à passer
- L'algorithme d'attente d'autorisation
  - attente active
- La libération pour autoriser d'autres processus



Conséquences

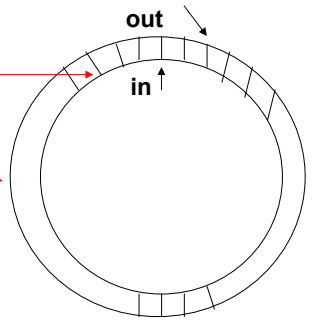
- Consomme des ressources (boucle d'attente)
- La famine est possible si un ensemble de tâches est bloquées, le déblocage n'est pas contrôlé
- L'interblocage est possible :
  - P1 exécute test_and_set
  - P1 est interrompu pour laisser l'UC à P2, plus prioritaire
  - P2 accède à la même section critique que P1 et exécute test_and_set et entre dans une attente active
  - P1 ne sera jamais reactivé => interblocage.
- Possibilité d'éviter l'attente active consommatrice de ressources par des appels systèmes SLEEP et WAKEUP pour suspendre l'appelant en attendant qu'une autre tâche le réveille

Exemple : un producteur consommateur

```
#define BUFFER_SIZE 10

typedef struct {
        ....
} item ;

item buffer [BUFFER_SIZE];
int in = 0 ;
int out = 0 ;
```



- in = = out => tampon vide
- ((in+1) % BUFFER_SIZE) == out => tampon plein

Le producteur

```
item nextProduced ;
while (true) {
    /* l'item à écrire est dans nextProduced */
    while (((in+1) % BUFFER_SIZE) == out) /* le buffer est plein  */
          ; /* rien à faire */
    buffer [in] = nextProduced ;
    in = (in+1) % BUFFER_SIZE ;
}
```

Le consommateur

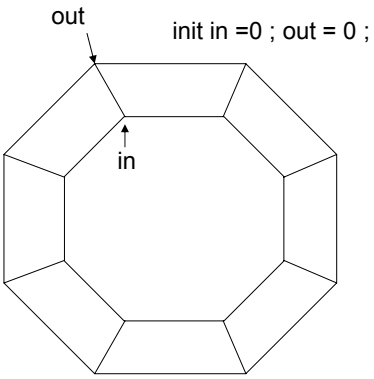
```
item nextConsumed ;
while (true) {
          while (in == out) /* le buffer est vide  */
          ; /* rien à faire */
    nextConsumed = buffer [out] ;
    out = (out+1) % BUFFER_SIZE ;
/* l'item est transféré dans nextConsumed */
}
```

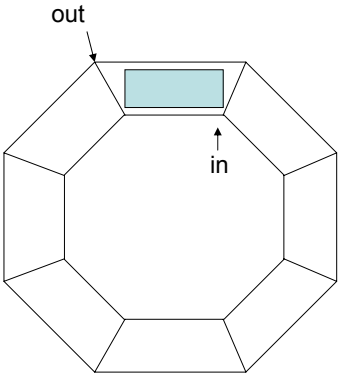

Programmation

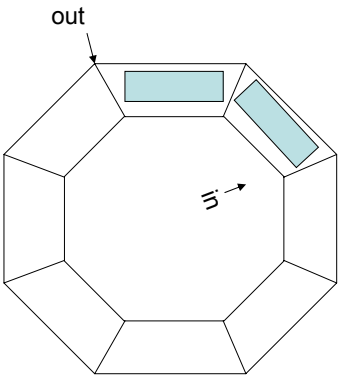
```
# include ''producteur_conso.h ''
# define N 10 ;        /* nb d'emplacements ds le tampon */
int in =out = 0 ;      /* nb d'objets ds le tampon : variable globale */
item buffer [BUFFER_SIZE];
```

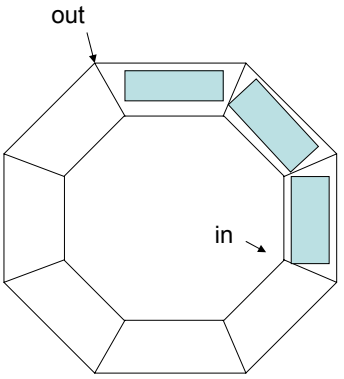
```
void producteur()
{
 item objet_produit ;
 while (TRUE)  {
          produire_objet(&objet_produit) ;
          if ((in+1)%BUFFER_SIZE == out) sleep () ;
          buffer[in] = objet_produit ;
          in = (in+1)%BUFFER_SIZE ;
          if (in == 1)
                wakeup(consommateur) ;
  }
}
```

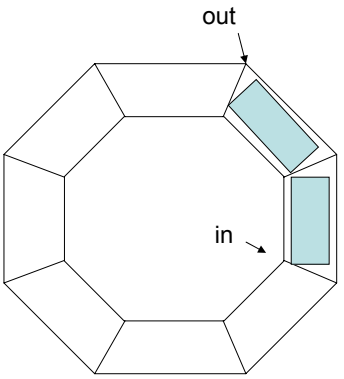
```
void consommateur()
{
 item objet_consomme ;
 while (TRUE)  {
       if (in == out) sleep () ;
        objet_consomme = buffer[out] ;
        out = (out+1)%BUFFER_SIZE ;
        if ( in == out) wakeup(producteur) ;
         consommer_objet(objet_consomme) ;
  }
}
```

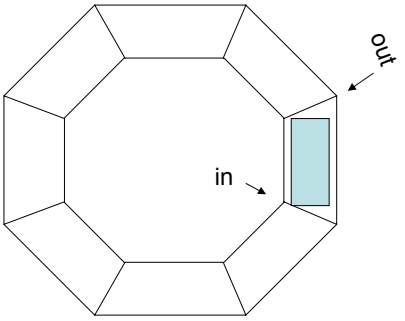

out
init in =0 ; out = 0 ;
in


out
in


out
in


out
in


out
in


out
in

Programmation

```
# include ‘’producteur_conso.h ’’
# define N 10 ;        /* nb d’emplacements ds le tampon */
int in =out = 0 ;      /* nb d’objets ds le tampon : variable globale */
item buffer [BUFFER_SIZE];
```

L’ordonnanceur décide de suspendre le consommateur et d’activer le producteur

```
void producteur()
{
 item objet_produit ;
 while (TRUE)  {
          produire_objet(&objet_produit) ;
          if ((in+1)%BUFFER_SIZE == out) sleep () ;
          buffer[in] = objet_produit ;
          in = (in+1)%BUFFER_SIZE ;
          if (in == 1)
                wakeup(consommateur) ;
  }
}
```

```
void consommateur()
{
 item objet_consomme ;
 while (TRUE)  {
       if (in == out) sleep () ;
        objet_consomme = buffer[out] ;
        out = (out+1)%BUFFER_SIZE ;
        if ( in == out) wakeup(producteur) ;
         consommer_objet(objet_consomme) ;
  }
}
```


- Le consommateur va exécuter l’instruction if et est suspendu par l’ordonnanceur : le tampon est vide et donc in = out = 0;
- Le consommateur suspendu, le producteur est élu ;
- in = 1 et out vaut toujours 0 ;
- Le consommateur est alors réveillé par wake_up bien qu’il n’ait jamais exécuté sleep ;
- Il poursuit son exécution avec in = 0 et donc exécute sleep ;
- Le producteur continue à remplir le tampon et se mettra en attente d’être réveillé par le consommateur qui vient d’exécuter sleep.

Interblocage

Les sémaphores

- Création d'une variable entière pour compter le nombre de variables en attente : le sémaphore.
- Le sémaphore vaut 0 si aucun réveil n'a été mémorisé, une valeur positive si un ou plusieurs réveils en attente ont été mémorisés.
- Le test du sémaphore, le changement de valeur et la mise en attente éventuelle de la tâche sont effectués en une seule opération atomique.
- Le sémaphore est accédé par deux opérations atomiques wait(S) et signal(S) initialement nommés P(S) et V(S).

Définitions

```
wait(S) {
    while S <= 0
            ;       /* rien */
    S -- ;
}

signal(S) {
    S ++ ;
}
```

Définitions (suite)

- Un sémaphore S est un élément de synchronisation auquel est associé :
  - Un compteur entier (S_CPT : nb réveils en attente)
  - Une file d'attente
  - Deux opérations atomiques (extension de sleep et wakeup) P(S) et V(S)
  - Une opération atomique d'initialisation E(S, valeur)
- Un sémaphore est dit binaire si S_CPT reste <= 1.

Sémantique

- P(S) ; opération atomique qui prend le sémaphore : bloquant si déjà pris
  - S_CPT - -
  - Si S_CPT < 0 alors
    - Bloquer l'appelant et mettre la tâche dans la file associée à S
  - Fin_Si
- V(S) ; opération atomique qui libère le sémaphore : jamais bloquant
  - S_CPT + +
  - Choisir une tâche de la file associée à S, la retirer et réveiller la tâche
  - Fin_Si
- E0(S,Valeur) ; opération atomique ; initialise S_CPT
  - S_CPT <- Valeur ;

Sémantique (suite)

- Le sémaphore possède la propriété suivante :
  - S_CPT < 0 => abs(S_CPT) == long(file) ;
- Pour un sémaphore binaire V(S) sans objet si S==1 ;
- Il n'est pas précisé quelle tâche est réveillée si plusieurs sont en attente :
  - FIFO
  - La plus prioritaire
- L'inversion de priorité reste possible

- Exemple 5 revu : ordonnancement des événements
  - *Soit y et z variables partagées ; y = z= 0;*

Tâche1 *Tâche2*

x = y+ z ; *y = 1 ;*

z = 2 ;

x peut valoir 0, 1, 2 ou 3.

Solution pour que x = 3

Soit s un sémaphore binaire déclaré :
binarySemaphore s ;
E0(s , 0) ;

Tâche1 *Tâche2*

s.P() ; *y=1 ;*
x = y+ z ; *z=2 ;*
s.V() ;

Le code de la tâche 2 sera exécuté **avant** le code de la tâche 1.

Exemple : un producteur consommateur revu

```
# include ''producteur_conso.h ''
# define BUFFER_SIZE 10                /* nb d'emplacements ds le tampon */
typedef int semaphore ;
semaphore vide ;
semaphore plein ;
item buffer [BUFFER_SIZE];

E0(vide, BUFFER_SIZE );
E0(plein, 0) ;
  void producteur()
  {
    item objet_produit ;
   int in = 0 ;
    while (TRUE)  {
              produire_objet(&objet_produit) ;
              P(&vide) ;
              buffer[in] = objet_produit ;
              in = (in +1)%n ;
              V(&plein) ;
    }
  }
```

```
void consommateur()
{
  item objet_consomme ;
  int out = 0 ;
  while (TRUE)  {
              P(&plein) ;
               objet_consomme = buffer[out] ;
               out = (out +1)%n ;
              V(&vide) ;
              consommer_objet(objet) ;
   }
}
```


Exemple d'implémentation en Java

```
synchronized public void P() {
            permit-- ;
            if (permit <0 )
               try { this.wait () ; }
            catch (InterruptedException e) { }
 }
```

```
synchronized public void V() {
            ++permit ;
            if (permit <= 0 )
               notify () ;
 }
```

suite