Théorie et pratique de la concurrence

Arnaud Sangnier

Étudiant : Colin González

Table des matières

1 Introduction **2**
1.1 Définitions . . . 3
1.1.1 État : . . . 3
1.1.2 Successeur : . . . 4
1.2 Problème de l'exclusion mutuelle . . . 4
1.2.1 Algorithme de Dekker . . . 8
1.2.2 Algorithme de Peterson . . . 12
1.2.3 Utilisation des verrous (*mutex*) . . . 14
1.3 Test-and-set et sémaphores . . . 18
1.3.1 Test-and-set . . . 18
1.3.2 Sémaphore . . . 19
1.3.3 Exemple : Producteur/Consommateur . . . 21
1.3.4 Le problème des philosophes . . . 23
1.3.5 Problème du bus . . . 25
1.3.6 Programmation concurrente en JAVA . . . 26
1.4 Algorithme de Friedberg et Peterson . . . 28
1.4.1 Sous-programme du "portier" . . . 30
1.5 Algorithmes distribués par partage de message . . . 30
1.5.1 Algorithme de Lamport . . . 32

Chapitre 1

Introduction

21 janvier 2016
Notation : contrôle continu (problème de concurrence sur papier) 1/4 + Examen 3/4
Notions de programmes concurrents :
Plusieurs programmes qui s'exécutent en même temps. Ces programmes peuvent communiquer entre eux. Soit par mémoire partagée ou par messages. Considérons trois programmes (exécutés par trois processus), P, Q et R. Les actions p_1, p_2, p_3, q_1, q_2, q_3 et r_1, r_2, r_3 sont réalisé de façon concurrentes sur un seul processus, les actions respectent l'ordre dans lequel elle sont données c'est-à-dire : pour P on à p_1 puis p_2 puis p_3. De plus on suppose que deux actions ne sont pas faite en même temps. Si on a une action par processus (p_1, q_1, r_1) : plusieurs séquences sont possibles :

p_1, q_1, r_1 ou q_1, p_1, r_1 ou r_1, q_1, r_1...

Si nous avons deux actions par processus :

$p_1, p_2, q_1, q_2, r_1, r_2$...

On appelle cela la sémantique d'entrelacement.
Prenons un programme séquentiel

```
int n=0;
int k1 = 1;
int k2 = 2; /* initialisation */
p1: n = k1;
p2: n = k2;
```

Pour représenter le comportement de ce programme on utilise un diagramme d'états. Un graphe où chaque noeud contient soit un pointer sur l'instruction soit une valeur pour chaque variable.
Pour les programmes concurrents :

```
int n = 0;
```

Process P	Process Q
`int k1=1;`	`int k2=2;`
`p1 : n = k1;`	`q1 n=k2;`

Problème de "data-race", selon l'exécution n peut prendre une valeur différente à la fin (ici 1 ou 2).
Remarque : on suppose que dans le cours que chaque instruction est daite de façon atomique (on n'a pas deux instructions qui s'exécutent en même temps et un instruction ne peux pas être arrêtée au milieu de son exécution).
Quand est-ce que cette hypothèse peut manquer de réalisme :

```
int n = 0;
```

Process P	Process Q
`p1 : n = n+1;`	`q1 n=n+1;`

Cependant si on oublie l'hypothèse d'atomicité, n peut valoir 1 ou 2 à la fin. P et Q accèdent sur une instruction en lecture et en écriture à la mémoire partagée, il se pet que ses accès ne soient pas atomiques. Si on veut observer ce phénomène avec l'hypothèse d'atomicité on peux modèliser les programme de la façon suivante.

```
int n = 0;
```

Process P	Process Q
`int tmp1=0;`	`int tnmp2=0;`
`p1: tmp1=n;`	`p2: tmp2=n;`
`p1 : n = tmp1+1;`	`q1 n = tmp2+1;`

A partir de maintenant toute instruction atomique (même celles qui lisent et écrivent en même temps)

1.1 Définitions

1.1.1 État :

l'état d'un programme concurrent P est un n-uplet contenant un pointeur d'instructions par procéssus et une valeur peur chaque variable (locale ou partagé)

1.1.2 Successeur :

Soient s_1 et s_2 deux états, alors s_2 est un successeur de s_1 (noté $s_1 \rightarrow_p s_2$) si s_2 correspond à la transformation de s_1 par l'application d'une instruction pointée dans s_1.
Le diagramme d'états de P est un graphe avec un somment initial $(S, \rightarrow_p s_0)$ où $s_0 \in S$ est l'état initial de P et S est le plus petit ensemble tel que si $s \in S$ et $s \rightarrow_p s'$ alors $s' \in S$

1.2 Problème de l'exclusion mutuelle

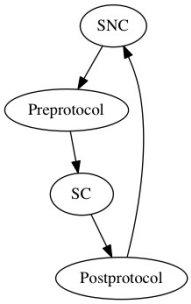
28 janvier 2016
Le problème $n \geq 2$ processus qui s'exécutent dans une boucle infinie.
Soient deux séquences d'instructions abstraites.

1. SNC : section no critique

2. SC : section critique

On doit fournir :
— une pré-protocol (demande d'accès à SC)

— un post-protocol (où on libère la SC)



Les propriétés de correction :

1. Exclusion mutuelle (jamais plus d'un processus en SC)

2. Absence d'interblocage (deux processus qui exécutent le pré-protocol ne se bloquent pas mutuellement)

3. Absence de famine (tout processus qui demande accès à la SC (par le pré-protocole) finit par entrer en SC)

4. Attente bornée (tout processus qui demande accès à la SC finit par entrer en un nombre borné d'étapes)

$4 \implies 3 \implies 2$
Hypothèse :
- le code a la forme (*)
- la SC ne bloque jamais
- ls SNC peut bloquer

Contexte : Communication par variable partagée. Les variables utilisées dans le pré et le post-protocole ne sont pas manipulées dans les sections SNC et SC.
Algo 1

```
int turn = 2;
```

```
while(true){
        SNC;                    /* p1 */
        while(turn != 1){}      /* p2 */
        SC;                     /* p3 */
        turn = 2;               /* p4 */
}
```

```
while(true){
        SNC;                    /* q1 */
        while(turn != 2){}      /* q2 */
        SC;                     /* q3 */
        turn = 1;               /* q4 */
}
```

Ici attente active sur la boucle while
Idée de l'algo : Si *turn* vaut 1, P a le droit d'accéder à la SC. Si *turn* vaut 2, Q a le droit d'accéder à la SC.
Propriété : L'algo 1 respecte la propriété de l'exclusion mutuelle.
Pour le montrer deux façons :

- Construire le diagramme d'états.
- Faire une preuve à la main.
- un état du digramme aura la forme (p_i, q_i, a) (a = turn)

— on peut "oublier" les variables de SNC et SC (qu l'on ne connaît pas). MAIS il ne faut pas oublier les hypothèses sur SNC et SC (par exemple SNC peut ne pas terminer)
— Pour vérifier l'exclusion mutuelle, il suffit de vérifier que (p_3,q_3, 1) et (p_3,q_3, 2) ne sont pas accessibles.

— Au plus 32 états ($4 * 4 * 2$) dans le diagrammes.

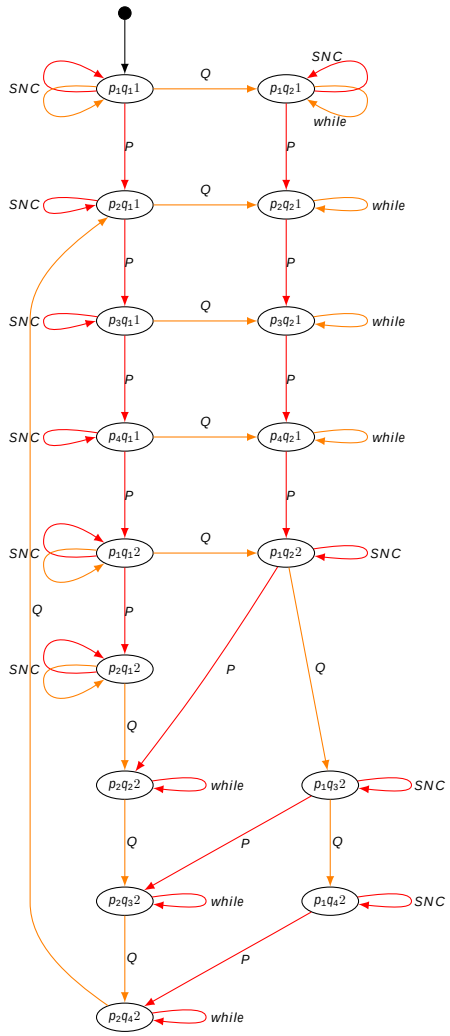


FIGURE 1.1 – Diagramme d'états

<u>L'exclusion mutuelle est vérifié :</u> car on a pas $(q_3, p_3, -)$ dans le diagramme d'états.
<u>Remarque :</u> prenons cette execution infinie :
$(p_1, q_1, 1) \rightarrow (p_1, q_2, 1) \leftarrow$ *while* ou le scheduler donne jamais la main

à P.

- famine car Q n'atteint jamais la section critique
- Execution non-réaliste
- On va considérer une hypothèse d'équité entre les processus

Hypothèse d'équité : Tout processus qui peut faire une action finira par la faire.

$\rightarrow (p_1, q_1, 1) \rightarrow_X (p_1, q_2, 1) \leftarrow_{while}$ pas équitable

$\rightarrow (p_1, q_1, 1) \rightarrow_{Equitable} (p_1, q_2, 1) \rightarrow_{SNC} (p_1, q_2, 1) \rightarrow_{while} (p_1, q_2, 1) \rightarrow_{SNC}$

Equitable on alterne SNC et while à l'infini

Dans le diagramme d'états on a plus de chemins que les exécutions "équitables de l'algo".

- Pour l'absence de famine on s'intéressera uniquement aux exécutions équitables.

Autre propriété : absence d'interblocage, on a au plus deux processus bloqués pour toujours dans le pré-protocole.Si P est en p_2 et Q en q_2 $turn$ vaut vaut 1 ou 2 et un des deux peut sortir.

Sous hypothèse d'équité, on a famine, prenons cette exécution équitable.

$\rightarrow (p_1, q_1, 1) \rightarrow_{Equitable} (p_1, q_2, 1) \rightarrow_{SNC} (p_1, q_2, 1) \rightarrow_{while} (p_1, q_2, 1) \rightarrow_{SNC}$

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
/* option -lpthread a la compilation */
#include <unistd.h>

void *f1( void *p);
void *f2( void *p);

void *f1(void *p){
        char *m =  (char *)p;
        /* sleep(temps_en_secondes): */
        /* int c = 4; */
        printf("%s\n", m);
        return NULL; /* pthread_exit(&c); */
        /* on passe 4 a la variable a mise dans le join*/
}

void *f2(void *p){
        char *m =  (char *)p;
        /* sleep(temps): */
        printf("%s\n", m);
        return NULL;
}

int main(int argc, char *argv[])
{
        pthread_t th1, th2;
        const char *s1 = "TH1";
        const char *s2 = "TH2";
         /* attribution de thread NULL option par defaut */
        int r1 = pthread_create(&th1,NULL, f1,(void *) s1);
        int r2 = pthread_create(&th2,NULL, f2,(void *) s2);
        /* int a; */
        pthread_join(th1, NULL); /* attend la fin de th1 */
        /* pthread_join(th1, &a); attend la fin de th1 */
        pthread_join(th2, NULL); /* attend la fin de th1 */

        return 0;

}
```

FIGURE 1.2 – POSIX Threads

1.2.1 Algorithme de Dekker

4 février 2016

```
boolean d1=false;
```

```
boolean d2=false;
int turn=1;
```

```
/* process p */
while(true){
        SNC;                          /* p1 */
        d1=true;                      /* p2 PP*/
        while(d2 == true){            /* p3 PP*/
                if(trun == 2){            /* p4 PP*/
                        d1=false;             /* p5 PP*/
                        while(turn !=1){} /* p6 PP*/
                        d1=true;              /* p7 PP */
                        }
        }
        SC;                           /* p8 */
        turn = 2;                     /* p9 */
        d1 = false;                   /* p10 */
}
```

```
/* process q */
while(true){
        SNC;                          /* q1 */
        d2=true;                      /* q2 PP*/
        while(d1 == true){            /* q3 PP*/
                if(trun == 1){            /* q4 PP*/
                        d2=false;             /* q5 PP*/
                        while(turn !=2){} /* q6 PP*/
                        d2=true;              /* q7 PP*/
                        }
        }
        SC;                           /* q8 */
        turn = 1;                     /* q9 */
        d2 = false;                   /* q10 */
}
```

<u>Idée :</u> Lorsque P veut accéder à sa SC

- il met $d1$ à $true$
- Si Q veut aussi accéder à sa SC la valeur de $turn$ détermine lequel des deux est prioritaire si c'est P alors Q attend sont tour et signale à P qu'il retire sa demande ($d2 = false$). Une fois que P sera passé, Q aura accès à sa SC.

P	Q	$d1$	$d2$	$turn$
p_1	q_1	$false$	$false$	1
p_2	-	-	-	-
p_3	-	$true$	-	-
p_3	q_2	$true$	$false$	1
p_8	-	-	-	-
-	q_3	$true$	$true$	-
-	q_4	-	-	-
-	q_5	-	-	-
-	q_6	-	$false$	-
-	q_6	-	-	-
p_8	q_6	$true$	$false$	1
p_9	-	-	-	-
p_10	-	-	-	2
-	p_7	-	-	-
p_10	q_3	$true$	$true$	2
-	p_4	-	-	-
-	p_3	-	-	-
-	-	-	-	-
p_1	p_3	$false$	$true$	2
-	p_3	-	-	-

TABLE 1.1 – Table d'états

Pour prouver ces algorithme, on va utiliser la notion d'invariants. Un invariant est une propriété qui est vrai à tout moment de l'exécution. Pour écrire nos invariants, on utilisera la grammaire suivant.

$I := I \vee I$
$|I \wedge I$
$|\neg I$
$|p_i$
$|q_j$
$|var == val$

FIGURE 1.3 – Grammaire des invariants

Exemple d'invariant :
Quand P est en p_8, $d1$ vaut $true$. $p_8 \implies d1 == true$
On va utiliser des invariants inductifs. I est un invariant inductif si est vrai dans l'état initiale du programme et si on prend un état (même non accessible) qui vérifie I alors tous ses successeurs vérifient I.
Exemple : $\neg(p_8 \wedge q_8)$ est un invariant mais ce n'est pas un invariant inductif.
$p_8, q_3, false, true, 1$ satisfait I mais

$p_8, q_3, false, true, 1 \rightarrow p_8, q_8, false, true, 1$
11 février 2016
Les trois propriétés suivantes sont des invariants.

1. $turn == 1 \vee turn == 2$
2. $p_3 \vee p_4 \vee p_5 \vee p_8 \vee p_9 \vee p_{10} \iff d1 == true$
3. $q_3 \vee q_4 \vee q_5 \vee q_8 \vee q_9 \vee q_{10} \iff d2 == true$

<u>Preuve :</u> 1. est un invariant inductif. Au début $turn = 1$, donc 1. est vraie.
Si on prend (p, q, t, a, b) qui vérifie 1. tous ses successeurs vérifient 1. (les seules instructions qui modifient $turn$ font $turn = 1$ ou $turn = 2$).
2. est vérifiée au début
Soit (p, q, t, a, b) un état qui vérifie 2.
Soit (p', q', t', a', b') un successeur de (p, q, t, a, b). Supposons que (p', q', t', a', b') ne vérifie pas 2. Alors on a $a' = false$ et $p' \in \{p_3, p_4, p_5, p_8, p_9, p_{10}\}$ ou $a' = true$ et $p' \in \{p_1, p_2, p_6, p_7\}$
Si $p = p'$ alors on a $a = a'$ car P est le seul à modifier $d1$. Donc on a $p \neq p'$. On fait ensuite une étude de cas (ou $p = p_1$).

- Si $p \in \{p_3, p_4, p_8, p_9\}$ alors $p' \in \{p_3, p_4, p_5, p_9, p_{10}\}$ et $a = a'$ vérifie 2. $\implies$ IMPOSSIBLE.
- Si $p \in \{p_5, p_{10}\}$ on a $a = true$ (par 2.) et $p' \in \{p_1, p_6\}$ et $a' = false$ impossible car $(p', q', t', a', b') \not\vdash 2$.
- Si $p \in \{p_1, p_2, p_6, p_7\}$ alors $a = false$ (car $(p, q, t, a, b) \vdash 2$.) et $p' \in \{p_2, p_3, p_7, p_8\}$ si $p' \in \{p_2, p_7\}$ alors $a' = false$ IMPOSSIBLE
- Si $p' \in \{p_2, p_3\}$ alors $a' = true \implies$ IMPOSSIBLE

Donc (p', q', t', a', b') vérifie 2., de plus 2. est un invariant inductif.
L'algorithme de Dekker respecte l'exclusion mutuelle.
<u>Preuve :</u> On raisonne par contradiction.
Supposons qu'on a $p_8 \wedge q_8$. Au coup d'avant on a soit $p_3 \wedge d2 == false \wedge q_8$ ou $q_3 \wedge d1 == false \wedge p_8$
Mais $q_8 \implies d2 == true$ (par invariant 3.) et $p_8 \implies d1 == true$ (par invariant 2.)
Donc on ne peut pas avoir $q_8 \wedge d2 == false$ ou $p_8 \wedge d1 == false \implies$ on a jamais $p_8 \wedge q_8$

18 février 2016
<u>Invariant inductif</u>

1. $turn = 1 \vee turn = 2$

2. $p_3 \vee p_4 \vee p_8 \vee p_9 \vee p_{10} \iff d1 = true$
3. $q_3 \vee q_4 \vee q_8 \vee q_9 \vee q_{10} \iff d1 = true$

L'algorithme de Dekker vérifie l'absence d'interblocage.
On raisonne par l'absurde en supposant qu'il y a interblocage alors on a au bout d'un moment toujours $(p_3 \vee p_4 \vee p_5 \vee p_6 \vee p_7) \wedge (q_3 \vee q_4 \vee q_5 \vee q_6 \vee q_7)$
Remarque : la valeur de $turn$ est alors fixée (les pré-protocoles ne modifient pas sa valeur). On suppose $turn = 1$ (le raisonnement est le même pour $turn = 2$) Donc on a au bout d'un moment que P est toujours soit en p_3 soit p_4. Par l'invariant 2. $d1$ reste à $true$. De plus régulièrement ($\neq$ toujours) $d2 = true$ sinon P ne serait pas bloqué en p_3 et p_4 (à chaque fois que P passe de p_3 à p_4, $d2 == true$).
Que peut faire Q :

- s'il est bloqué en q_6, impossible sinon $d2$ vaut toujours $false$.
- donc il boucle en q_3 mais comme $turn = 1$, il va se retrouver bloqué en $q_6 \implies$ contradiction

Ceci veut dire qu'on n'a pas interblocage.
On peut montrer que l'algorithme de Dekker respecte l'absence de famine (et même l'attente bornée).

1.2.2 Algorithme de Peterson

```
boolean d1=false;
boolean d2=false;
int turn=1;
```

```
/* process p */
while(true){
        SNC;                          /* p1 */
        d1=true;                      /* p2 PP*/
        turn = 2;                     /* p3 PP*/
        while(d2 == true && turn == 2){} /* p4 PP*/
        SC;                           /* p5 */
        d1 = false;                   /* p6 */
}
```

```
/* process q */
while(true){
        SNC;                          /* q1 */
        d2=true;                      /* q2 PP*/
        turn = 1;                     /* q3 PP*/
        while(d1 == true && turn == 1){} /* q4 PP*/
        SC;                           /* q5 */
        d2 = false;                   /* q6 */
}
```

Idée : lorsque P veut accéder à sa SC

1. il signale en mettant $d1$ à $true$
2. donne la priorité à Q (en mettant $turn$ à 2)
3. Il vérifie que Q ne veut pas y aller ($d2 = false$) ou qu'il est prioritaire ($turn == 1$)

L'algorithme de Peterson vérifie l'exclusion mutuelle.
Preuve : Raisonnement par l'absurde que P soit en SC lorsque Q y arrive. On a alors $d1 = d2 = true$ Comme Q arrive à la SC on a $turn = 2$. Donc P a exécuté $turn = 2$ après que Q n'exécute $turn = 1$ Lorsque P rentre en SC et $turn = 2$ et $d2 == true \implies$ contradiction ce qui garantie l'exclusion mutuelle.
L'algorithme de Peterson respecte l'absence d'interblocage. Preuve : s'il y a interblocage P et Q sont bloqué pour toujours en p_4 et q_4 mais alors on a $d2 == true \wedge d1 == true \wedge turn == 1 \wedge turn == 2 \implies$ IMPOSSIBLE.
L'algorithme de Peterson respecte l'absence de famine.
Preuve : Supposons famine pour P Donc P est bloqué en p_4 et donc on a $d2 == true$ et $turn == 2$ régulièrement et en plus $d1 == true$ toujours.
Quand $d2 == true$. Q est q_3, q_4, q_5 ou q_6

- S'il est en q_3, il mettra $turn$ à 1 et P passera en SC $\rightarrow$ impossible.
- Si Q est q_4, comme $d2 = true$ et $turn = 2$ régulièrement Q

passera en SC puis en q_1 et retournera en q_3 (car $d2$ vaut $true$ régulièrement) $\Longrightarrow$ (cf. premier cas) IMPOSSIBLE.

Ceci prouve qu'on a absence de famine (en fait attente bornée).

1.2.3 Utilisation des verrous ($mutex$)

- on va utiliser la libraire $pthread.h$
- types des verrous $pthread_mutex_t$
- création des verrous en C statiquement

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

- dynamiquement

```
int pthread_mutex_int(pthread_mutex_t *mutex, const pthread_mu
/* attributs=NULL pour les options par defaut*/
```

- destruction de verrous (dans le cas de création dynamique)

```
int pthread_mutex_destroy(pthread_mutext_t *m);
```

  détruit uniquement si m est déverrouillé.
- prendre le verrou

```
int pthread_mutex_lock(pthread_mutex_t *m)
```

  bloquant si le verrou est pris
- libérer le verrou

```
int pthread_mutex_unlock(pthread_mutex_t *m)
```

- tester si un verrou est pris et le prendre le cas échéant

```
int pthread_mutex_trylock(pthread_mutex_t *m)
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

typedef struct _concur_x {
        int x;
        pthread_mutex_t *m;
} *concur_x;

concur_x compteur;

void *test_et_inc(void *p){
        pthread_mutex_lock(compteur->m);
        if(compteur->x < 1){
                compteur->x++;
        }
        pthread_mutex_unlock(compteur->m);
        return NULL;
}

int main(int argc, char *argv[])
{
        compteur = (concur_x) malloc(sizeof(struct _concur_x*));
        pthread_mutex_init(compteur->m, NULL);
        compteur->x = 0;
        pthread_t t1, t2;
        pthread_create(&t1,NULL, test_et_inc,NULL);
        pthread_create(&t2,NULL, test_et_inc,NULL);
        pthread_join(t1,NULL);
        pthread_join(t2,NULL);
        printf("%d\n", compteur->x);
        return 0;
}
```

FIGURE 1.4 – Exemple mutex

Problème de l'attente active :

Thread 1	`lock(mutex);`
	`while(flag != 1){`
	`}`
	`unlock(mutex);`

Thread 2	`lock(mutex);`
	`flag = 1;`
	`unlock(mutex);`

Variables conditions :

Type :

```
pthread_cond_t vc;
```

Initialisation :

```
pthread_t_cond_init(&vc,NULL);
```

Attente :

```
pthread_cond_wait(&vc, &mutex);
```

Libère le mutex et endort le thread, au réveil, on récupère le verrou.
Réveil :

```
pthread_cond_signal(&vc);
```

Réveille un thread endormis sur vc si aucun thread ne dort, il ne fait rien.

```
pthread_cond_broadcast(&vc);
```

Réveille tous les threads endormis sur vc

- Si un thread est réveille et le verrou est pris il attend la libération du verrou (il ne se rendort pas sur vc)
- pas le même thread

Condition :

```
lock(mutex);
while(flag != 1){
        wait(&vc,&mutex);
}
unlock(mutex);
```

```
        lock(mutex);
        flag = 1;
        signal(&vc);
        unlock(mutex);
```

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

typedef struct _liste
{
        int val;
        struct _liste next;
} liste;

liste l = NULL;

int nb_elem = 0;

pthread_mutex_t mut;
pthread_cond_t full;
pthread_cond_t empty;

void cons (void p)
{
        pthread_mutex_lock (&mut);
        while ( nb_elem == 0 )
        {
                pthread_cond_wait (&empty, &mut);
        }
        printf ("val %d\n", l >val);
        liste tmp = l;
        l = l >next;
        free (tmp);
        if (nb_elem == 10)
        {
                pthread_cond_broadcast (&full);
        }
        nb_elem ;
        pthread_mutex_unlock (&mut);
        return NULL;
}

void prod (void p)
{
        pthread_mutex_lock (&mut);
        while (nb_elem == 10)
        {
                pthread_cond_wait (&full, &mut);
        }
        liste tmp = (liste) malloc ( sizeof (struct _liste));
        tmp >val = random () & 99;
        tmp >next = l;
        l = tmp;
        if (nb_elem == 0)
        {
                pthread_cond_broadcast (&empty);
        }
        nb_elem++;
        pthread_mutex_unlock (&mut);
        return NULL;
}
```

valeur initiale du sémaphore. On a toujours

1. $S.V \geq 0$
2. $S.V = k + Nb(signal(s)) - Nb(wait(s))$

$Nb(signa(s))$ le nombre de *signals* effectué sur s (signifie que l'on est sorti).
$Nb(wait(s))$ le nombre de *wait* effectué sur s (signifie que l'on est sorti).
L'algorithme respecte l'exclusion mutuelle.
Preuve : Soit N le nombre de processus en SC. On a alors $N = Nb(wait(s)) - Nb(signal(s))$ De plus, a tout moment on a $S.V = 1 + Nb(signal(s)) - Nb(wait(s)) = 1 - N$. Donc comme $S.V \geq 0$ on a $1 \geq N$. Le nombre de processus en SC est toujours ≥ 1.
L'algorithme respecte l'absence d'interblocage. Si on a $N = 0$. Cependant si $S.V = 1$ et alors la seul instruction bloquante *wait* ne peut pas bloquer.
L'algorithme respecte l'absence de famine. Si P est bloquée sur le *wait*, alors $S.V = 0$ est en SC et donc en sortant il réveillera P.
Example d'utilisation de sémaphore Pour résoudre un problème de précédence.
Deux tâches T1 et T2 exécutées par deux processus différents on veut être sur que T2 soit fait après T1. On initialise le sémaphore à 0 après T1 on fait *signal* et avant T2 on fait *wait*.

1.3.3 Exemple : Producteur/Consommateur

- les producteurs écrivent des données dans une FIFO
- les consommateurs consomment les données et les retirent de la structure à la consommation.

1. La structure est infinie, on ne veut pas d'attente active des consommateurs.
2. la structure est de taille finie N, on ne veut pas d'attente active chez les consommateurs ou les producteurs (structure vide ou pleine respectivement).

Structure infinie

Données partagées

```
StructureInfinie F = [];
Semaphore NonVide = (0,[]);/** valeur initiale liste des process en attente */
```

Process Producteur	`while ( true )`
	`{`
	`data d = production(); /** q1 */`
	`ajouter (d,F); /** q2 */`
	`signal (NonVide); /** q3 */`
	`}`

Process Consommateur	`while ( true )`
	`{`
	`wait (NonVide); /** q1 */`
	`d = extraire(F); /** q2 */`
	`consommer (d); /** q3 */`
	`}`

FIGURE 1.10 – Producteurs consommateurs (sans attente active) FIFO infinie

Structure finie

Données partagées

```
StructureFinie F = [N];
Semaphore NonVide = (0,[]);/** valeur initiale liste des process en attente */
Semaphore NonPlein = (n,[]);
```

Process Producteur	`while ( true )`
	`{`
	`data d = production(); /** q1 */`
	`wait (NonPlein); /** q2 */`
	`ajouter (d,F); /** q3 */`
	`signal (NonVide); /** q4 */`
	`}`

Process Consommateur	`while ( true )`
	`{`
	`wait (NonVide); /** q1 */`
	`d = extraire(F); /** q2 */`
	`signal (NonPlein ) /** q3 */`
	`consommer (d); /** q4 */`
	`}`

FIGURE 1.11 – Producteurs consommateurs (sans attente active) FIFO finie

Remarque : On peut proteger les lignes p_2 et q_2 par une sémaphore binaire (si les opérations ajouter et extraire ne sont pas faites

de façon atomique).
Intuition : Il y a un lien entre le nombre de données dans la F et la valeur de non vide. On a exactement que $NonVide.v \simeq |F|$ où $|F|$ = le nombre de données.

1.3.4 Le problème des philosophes

Quand le philosophe a pris une baguette, tant qu'il ne l'a pas libérer elle n'est pas disponible pour un autre. Les philosophes font en boucle :

— penser
— prendre les baguettes
— manger
— laisser les baguettes

Les propriétés :

1. on ne mange que si l'on a ses deux baguettes.
2. une baguette n'est tenue que par un philosophe.
3. absence d'interblocage.
4. absence de famine

```
Semaphore[] baguettes = [(1,[]),...,(1,[])]; /** 0 - n-1*/
/** process philosophe */
while (true)
{
        penser;
        wait (Baguette[i]); /** prend la baguette de gauche */
        wait (Baguette[(i+1)% n]); /** prend la baguette de droite */
        manger;
        signal (Baguette[i]); /** rend la baguette gauche */
        signal (Baguette[(i+1) % n]); /** rend la baguette droite */
}
```

FIGURE 1.12 – Solution facile

Interblocage : tous les philosophes prennent leur baguette gauche et ils sont ensuite tous bloqués en attente de leur baguette droite. 1. et 2. sont satisfaites par cette solution 3. ne l'est pas.
10 mars 2016
Autre solution :
On disting un philosophe qui ne prend pas ses baguettes dans le même ordre que les autres.

```
Semaphore[] Baguette[(1,[]),...(1,[])]; /** n baguettes */
```

Philosophe 0	`while(true)`
	`{`
	`penser();` `/** p0,1 */`
	`wait(Baguette[1]);` `/** p0,2 */`
	`wait(Baguette[0]);` `/** p0,3 */`
	`manger();` `/** p0,4 */`
	`signal(Baguette[1]);` `/** p0,5 */`
	`signal(Baguette[0]);` `/** p0,6 */`
	`}`

Philosophe i $i \in [1;:::;n-1]$	`while(true)`
	`{`
	`penser();` `/** pi,1 */`
	`wait(Baguette[i]);` `/** pi,2 */`
	`wait(Baguette[(i+1)%n]);` `/** pi,3 */`
	`manger();` `/** pi,4 */`
	`signal(Baguette[(i+1)%n]);` `/** pi,5 */`
	`signal(Baguette[0]);` `/** pi,6 */`
	`}`

FIGURE 1.13 – Solution au problème des philosophe.

1. est vérfiée (on ne mange que si l'on a les deux baguettes). 2. est vérifiée (une baguette n'est tenue que par un philosophe). 3. est vérifiée.
Le philosophe 1 est aussi bloqué en $p_{i,3}$ et il a pris la baguette $(i+1)$. Le philosophe $(n-2)$ est bloqué en $(p_{n-2,3})$. Le philosophe $(n-1)$ attend la baguette 0 $\implies$ impossible. car dans ce cas $Philo[0]$ mange. Si il y a interblocage, tous les philosophes sont bloqués soit en $p_{i,2}$ soit en $p_{i,3}$. Si un philosophe est bloqué en $p_{i,3}$ si $i = 0$, alors Philosophe n-1 mange il y a donc contradiction. Si $i \in [1, ..., n-1]$ alors, si il y a interblocage il sont tous bloqués en $p_{i,2}$. Cependant 0 est bloqué en $p_{0,2}$ et 1 est bloqué en $p_{1,2}$. Ceci est impossible, un des deux doit pouvoir prendre la baguette 1. $\implies$ Absence d'interblocage.
<u>Absence de famine :</u>
un processus en attente sur une baguette va finir par être réveillé. Comme l est seul en attente, il va pouvoir prendre la baguette. Marche avec des sémaphores faibles.
<u>Autre solution</u> On autorise que $(n-1)$ philosophe à manger en même temps. On rajoute un sémaphore initiale à $(n-1)$.

```
Sempahore[] Salle=[(1,[])];
Semaphore[] baguettes = [(1,[]),...,(1,[])]; /** 0 - n-1*/
/** process philosophe */
while (true)
{
        penser;
        wait (Salle);
        wait (Baguette[i]); /** prend la baguette de gauche */
        wait (Baguette[(i+1)% n]); /** prend la baguette de droite */
        manger;
        signal (Baguette[i]); /** rend la baguette gauche */
        signal (Baguette[(i+1) % n]); /** rend la baguette droite */
        signal (Salle);
}
```

FIGURE 1.14 – Autre solution avec semaphore Salle

1.3.5 Problème du bus

On a un processus bus de capacité $K < N$. En boucle, il attend d'être plein, puis il part, ensuite il attend d'être vide. On a N passagers. En boucle, il essaient de monter dans le bus, ils sont dans le bus et ils descendent.

```
Semaphore Plein = (0,[]);
Semaphore DescendreBus = (0,[]);
Semaphore Vide = (0,[]);
Semaphore CapaBus = (k,[]);
Semaphore Verrou = (1,[]);
int cpt = 0;
```

Processus bus	`while(true)`
	`{`
	`wait(plein);`
	`roule();`
	`for(int i=0; i<k;i++)`
	`{`
	`signal (DescendreBus);`
	`}`
	`wait (vide);`
	`}`

Processus passager	`while(true)`
	`{`
	`wait (capaBus);`
	`wait (Verrou);`
	`cpt = cpt + 1;`
	`if (cpt = k)`
	`{`
	`signal (Plein);`
	`}`
	`signal(Verrou);`
	`wait (DescendreBus);`
	`wait (Verrou);`
	`cpt = cpt-1;`
	`if (cpt == 0)`
	`{`
	`signal (Vide);`
	`for (int i=0;i<k;i++)`
	`{`
	`signal (CapaBus);`
	`}`
	`}`
	`signal (Verrou);`
	`}`

FIGURE 1.15 – Solution au problème des philosophe.

1.3.6 Programmation concurrente en JAVA

Tout objet en JAVA est équipé d'un verrou, pour rendre atomique la manipulation d'un objet, on utilise le mot clef *synchronized*. Il y

a deux méthodes.

1. étendre la classe Thread et redéfinir la méthode run.
2. implémenter l'interface Runnable, écrire la méthode run.

```
public class MonCode implements Runnable{
    Object a;
    Public MonCode(Object a){
        this.a = a;
    }
    public static void run(){
        /** code a executer */
    }
    public static void main (String [] args){
        Object b;
        Thread t=new Thread(new MonCode(b));
        t.start();
    }
}

public class SharedInt{
    private int value=0;
    public synchronized void inc() {
        value=value+1;
    }
    /** notify(): reveiller un processus bloque sur cet objet. */
    /** notifyAll(): reveiller tous les processus. */
}

public class MonCode implements Runnable{
    SharedInt a;
    public MonCode(SharedInt a){
        this.a = a;
    }
    public static void run(){
        a.inc();
    }
    public static void main (String [] args){
        SharedInt i= new SharedInt();
        Thread t1=new Thread(new MonCode(i));
        Thread t2=new Thread(new MonCode(i));
        t.start();
    }
    public synchrnonized void dec(){
```

```
        while (value <=0){
            wait(); /** endort le thread et libere le verrou */
        }
        value = value -1;
    }
}

public class Semaphore {
    private int v;
    private int dort;
    public Semaphore (int v) {
        this.v=v;
    }
    public synchronized void sem_wait (){
        if (v==0){
            dort++;
            wait();
            dort--;
        } else {
            v=v-1;
        }
    }
    public synchronized void sem_signal(){
        if (v==0){
            v=v+1;
        } else {
            notify();
            dort--;
        }
}
```

1.4 Algorithme de Friedberg et Peterson

Algorithmes d'exclusion mutuelle. Pour $n \geq 2$ processus utilisant deux sémaphores faibles et deux variables partagées.

```
boolean vide = False;
int num = 0;
SemaphoreFaible S[0]=(1,[]),S[1] = (1,[]);
```

Processus P	`while (true)`
	`{`
	`SNC;`
	`int nomnum = num;`
	`wait(S[monnum]);`
	`if (num == monnum)`
	`{`
	`int autrenum = 1 - monnum;`
	`wait(S[autrenum]);`
	`num = autrenum;`
	`do`
	`{`
	`vide = true;`
	`signal (S[monnum]);`
	`wait (S[monnum]);`
	`}while (vide = false);`
	`SC;`
	`signal (S[autremnum]);`
	`}`
	`else`
	`{`
	`vide = false;`
	`SC;`
	`}`
	`signal (S[monnum]);`
	`}`

FIGURE 1.16 – Algorithme de Friedberg et Peterson

Le premier processus qui fait le if change le num et garantit que :

- tous les processus bloqués sur $S[monnum]$ passe en SC.
- lui il passe en SC
- reveille un processus bloqué sur $S[1 - monnum]$ et ainsi de suite.

1.4.1 Sous-programme du "portier"

```
boolean Vide = false;
Semaphore faible S = (0,[]);
```

Processus P	wait(S);
	Vide = False;
	signal(S);

Processus Portier	**do** { vide = **true**; signal (S); wait (S); }**while** (vide== **false**);

FIGURE 1.17 – Sous-programme du "portier".

Le processus portier sort de sa boucle que si il y a personne (aucun P_i). Bloqué en $wait(S)$.

1.5 Algorithmes distribués par partage de message

Les processus communiquent maintenant par envoi et réception de messages. Les processus sont distribués sur différentes machines (appelés sites).

— chaque site a sa mémoire locale
— chaque site peut recevoir des messages des autres sites
— sur chaque site, les processus sont divisées en 2 parties

1. partie chargée de l'execution du code du processus
2. partie chargée de la réception des messages

Les deux parties s'exécutent en parallèle et accèdent toutes deux à la mémoire partagée.

Hypothèse :

— un site ne peut pas s'arrêter. Il peut toujours recevoir et envoyer des messages.
— Chaque site peut communiquer avec les autres sites.
— Les canaux sont fiables $\Longrightarrow$ aucun message n'est perdu ou détérioré.
— Le temps de communication est arbitraire (délai entre émission et réception d'un message) mais fini.

On notera $send(m) \rightarrow P$ l'envoi du message m à P.

Processus P1	`boolean askSC = false;`
	`boolean inSC = false;`
	`while (true)`
	`{`
	`  SNC;`
	`  askSC = true;`
	`  while (inSC == false)`
	`  {`
	`  send (req1) -> P2;`
	`  }`
	`  SC;`
	`  inSC = false;`
	`  askSC = false;`
	`}`
	`/** gestion de message de p1 */`
	`case (req2):`
	`  if (askSC == False)`
	`  {`
	`  send (ok1) -> P2`
	`  } /** execution atomique */`
	`case (ok2):`
	`  inSC == true; /** code atomique */`

Processus P2	`boolean askSC = false;`
	`boolean inSC = false;`
	`while (true)`
	`{`
	`  SNC;`
	`  askSC = true;`
	`  while (inSC == false)`
	`  {`
	`  send (req2) -> P1;`
	`  }`
	`  SC;`
	`  inSC = false;`
	`  askSC = false;`
	`}`
	`/** gestion de message de p2 */`
	`case (req1):`
	`  if (askSC == False)`
	`  {`
	`  send (ok2) -> P1`
	`  } /** execution atomique */`
	`case (ok1):`
	`  inSC == true; /** code atomique */`

FIGURE 1.18 – Synchronisation par canaux.

Cet algorithme ne respecte pas l'exclusion mutuelle.
Exemple d'exécution $P1$ envoie deux ($req1$) reçoit deux ($ok2$) prend le premier va en SC sort de SC $P1$ envoie un ($req2$) reçoit ($ok1$) va en SC $P1$ prend le deuxième ($ok2$) va en SC.

1.5.1 Algorithme de Lamport

24 mars 2016
Hypothèse : les messages ne peuvent pas se doubler.

- si le processus P_i envoie M_1 au processus P_j puis encore M_2 au processus P_j alors le processus P_j recevra M_1 puis M_2.

Idée de l'algorithme :

- n processus qui veulent accéder à une section critique SC.
- Chaque processus dispose d'une *horloge* discrète. C_i (un entier positif).
- Chaque processus dispose des valeurs d'horloge de tous les autres voulant accéder à SC et il accède à sa SC si sa valeur d'horloge et la plus ancienne (sa valeur est la plus petite que les autres).

Un processus libérant sa SC envoie un message *release* pour dire aux autres processus de ne plus prendre en compte son horloge.
Tout processus demande à recevoir un message *recent* de la part des autres processus avant d'accéder à sa SC.
Chaque processus envoie donc un message ACK (accusé de réception) à chaque fois qu'il reçoit une requête.
Fonctionnement des horloges : chaque processus fait $+1$ sur son horloge. Quand un processus reçoit une message daté t et $C_1 < t$ alors C_1 devient t. Pour gérer les horloges avec la même valeur, on définit l'ordre $<< (a, b) << (a', b') \iff (a < a')$ ou $(a = a')$ et $(b < b')$ ou a est l'horloge et b le numéro de processus.
Description des messages : Ils sont de la forme (m, t, k) où m est le type du message $(request, release, ack)$, t la valeur de l'horloge et k le numéro du processus $\in \{1, ..., n\}$. De plus, $date(m, t, k) = (t, k)$.
Chaque processus a un tableau de taille n initialisé de la façon suivante $\forall i \in \{1, ..., n\}$ $Mess[i] = (ack, -1, 0)$. Ce tableau set à stocker les derniers messages reçus. Le parties entre $[-]$ sont exécutée de façon atomique.

Processus P_i	`Message[] Mess = {(ack, -1,0), (ack,-1,0),...,(ack,-1,0)};`
	`int c=0;`
	`while(true)`
	`{`
	`SNC /** p1 */`
	`[for all j!= i do /** p2 */`
	`{`
	`send(req,c,i);`
	`}`
	`Mess[i] = (req,c,i); /** p3 */`
	`c=c+1;] /** p4 */`
	`while ( exists j != i, date(M[j]) << date(M[i]) /** p5 */`
	`{}`
	`SC /** p6 */`
	`/** M[i] est le message le plus ancien de Mess */`
	`[ forall j !=i do /** p7 */`
	`{`
	`send(release, c, i) -> Pj;`
	`}`
	`Mess[i] = (release, c,i); /** p8 */`
	`c = c+1;] /** p9 */`
	`}`

FIGURE 1.19 – Algorithme de Lamport

```
case (req,t,j):
        [if (t>c)
        {
                c=t
        }
        c = c+1;
        Mess[j] = (req,t,j);
        Send(ack,c,i)->Pj;]
case (release,t,j):
        [if (t>c)
        {
                c=t
        }
        c=c+1;
        Mess[j] = (release,t,j);]
case (ack, t, j):
        [if (t>c)
        {
                c=t
        }
        c= c+1;
        if(type(Mess[j] = req)
        {
                Mess[j]=(ack,t,j);
        }]
```

FIGURE 1.20 – Reception des messages de Pi

Montrons alors que l'algorithme respecte l'exclusion mutuelle. Supposons que P_i et P_j soient en même temps en p_6. Soit $Mess^i$ le tableau de P_i à ce moment et $Mess^j$ le tableau de P_j. On a $Mess^i[i] = (req, t, i)$ et $Mess^j[j] = (req, t', j)$, supposons que $(t, i) << (t', j), (i < j)$. Si P_j a accédé à sa SC, il a un message récent de i dans $Mess^j[i]$. Si ce message est (req, t'', i) ou $(release, t'', i)$ avec $(t, j) << (t'', i)$ alors on a $t < t''$ et donc $t' < t''$ on a plus dans $Mess^i[i]$ le message $(req, t, i) \implies$ impossible.
Si dans $Mess^j[i] = (ack, t'', i)$ alors $t'' < t$ donc ack a été envoyé après (req, t, i) donc P_j a reçu (req, t, i) pour (ack, t', i) si P_j a écrit (ack, t'', i) dans $Mess^j[i]$ entre $t < t''$. P_i a envoyé $(release, t''', i)$ avec $t < t''' < t''$ et donc on a plus $Mess^i[i] = (req, t, i)$
31 mars 2016

Algorithme de Ricart-Agrawala (1981)

<u>Idée :</u> Un processus P_i communique à tous les autres qu'il souhaite accéder à sa SC.

— Un autre processus P_j répond OK si il ne souhaite pas accéder à sa SC s'il n'est pas prioritaire.
— Si P_j ne répond pas tout de suite, il le fera après avoir accédé à sa SC.
— chaque processus utilise un tableau $Waiting$ pour stocker la liste des processus à qui il doit envoyer OK.
— le priorité est établie pas un numéro que chaque processus choisit avant d'accéder à sa SC (plus le numéro est petit, plus un processus est prioritaire).
— le numéro est choisi plus grand que tous les numéros que le processus a vu.
— un processus accède pa sa SC lorsqu'il a reçu OK de tout le monde.

N.B. : Il n'y a pas d'hypothèse sur l'ordre des messages. Deux messages envoyés par P_i à P_j n'arrivent pas en même temps à P_j.

Processus P_i

```
boolean[n] Waiting = {false,...,false}; /** n fois */
boolean reqCS = False;
int pr = 0;
int maxpr = 0;
int nbok = 0;
while(true)
{
        SNC /** p1 */
        [recCS = true;
        nbok = 1;
        pr = prmax +1 ;]
        forall j != i{
                send(req,pr,i) -> Pj
        }
        while(nbok > 0) {}
        SC;
        [reqCS = False;
        forall j in {1,...,n}{
                if (Waiting[j] == True)
                {
                        Waiting[j] = False;
                        send(ok, i) -> Pj;
                }
        }]
}
```

FIGURE 1.21 – Algorithme de Ricart-Agrawala

```
case (ok,i):
        nbok = nbok-1;
case (req,k,j):
        [if(k > maxpr) { maxpr = k}
        if ( reqCS == False || (k,j) << (pr,i)
        {
                send(ok,i) -> pj;
        }
        else
        {
                Waiting[j] = True;
        }
        ]
```

FIGURE 1.22 – Reception des messages de Pi

Théorème L'algorithme de Ricart-Agrawala respecte l'exclusion mutuelle l'absences de famine (et donc l'absence d'interblocage).

Algorithme pour n processus avec varibales propres

- chaque variable partagée peut être lue par tous mais n'est modifiée que par un seul (multiple reader / single writer).
- algorithme de Lamport (1974)
- algo classique utilisé dans les files d'attente
- algo de la boulangerie vérifie l'exclusion mutuelle et l'absence de famine, attention les valeurs des entiers dans nb peuvent être non-bornées.

Processus P_i

```
boolean[n] chosing = {false,...,false}; /** n fois */
int[n] Nb = {0,...,0}; /** n fois */
while(true)
{
        SNC /** p1 */
        choosing[i] = True; /** violet */
        Nb[i] = max(Nb) + 1;
        choosing[i] = false; /** violet */
        for (j = 1, j <= n, j++)
        {
                if(j != i)
                {
                        while(choosing[j] == True){}; /** violet */
                        while(Nb[j]==0 && (Nb[i],i) << (Nb[j],j))
                        {};
                }
        }
        SC
        Nb[i] = 0;
}
```

FIGURE 1.23 – Algorithme de la boulangerie

Si on enlève la section violette, il n'y a pas d'exclusion mutuelle.

— processus 2 fait le calcul de Max en même temps que processus 1 mais il termine avant et après que processus 1 ait le $Nb[2]$. Il a comme valeur 1. Il fait la boucle for avant que P_1 mette à jour Nb[1] et passe en SC $Nb[1] = 0$. Ensuite P1 met a jour $Nb[1]$ a 1 et rejoint P_2 en SC.

•