Exercice 1 :

Rendez vous à N processus Soient N processus parallèles ayant un point de rendez-vous. Un processus arrivant au point de rendez-vous se met en attente s'il existe au moins un autre processus qui n'y est pas arrivé. Le dernier arrivé réveillera les processus bloqués. Proposer une solution qui permet de résoudre ce problème en utilisant des sémaphores.

```
sémaphore mutex = 1, s= 0;
entier NbArrivés = 0; /*nbre de processus arrivés au rendez-vous*/
Procédure RDV
Début
down(mutex);
NbArrivés = NbArrivés + 1;
Si (NBArrivés < N) Alors /* non tous arrivés */
up(mutex); /* on libère mutex et */
down(s); /* on se bloque */
Sinon
up(mutex); /* le dernier arrivé libère mutex et */
Pour i = 1 à N-1 Faire
up(s); /* réveille les N-1 bloqués */
Finsi
Fin
```

Exercice 2 : Gestion du trafic aérien

Le but de cet exercice est la gestion du trafic aérien. On ne dispose que d'une seule piste à la fois d'atterrissage et de décollage. En plus, cette piste ne peut accepter qu'un seul avion quelque soit la manœuvre (atterrissage ou décollage). Pour cela, on dispose de deux files d'attente :

- en l'air de taille N pour les avions souhaitant atterrir

- et au sol de taille M pour les avions souhaitant décoller.

La gestion de ce trafic d'avions nécessite, alors, quatre fonctions :

- une fonction SortirAvions ( ) qui fait sortir les avions du hangar (dépôt) et les place dans la file d'attente de décollage,
- une fonction Decollage ( ) qui prend un avion cloué en sol dans la file d'attente de décollage et le fait décoller en utilisant la piste,
- une fonction AmenerAvions ( ) qui fait entrer, dans la file d'attente d'atterrissage, des avions en vol,
- et une fonction Atterrissage ( ) qui prend un avion de la file d'attente d'atterrissage et le fait atterrir en utilisant la piste.

```
#define N 5
 #define M 5
 semaphore Decollage_vide, Decollage_plein, Atterrissage_vide, Atterrissage_plein, Piste;
Sem_Init (Decollage_vide, M); //initialisation du sémaphore Decollage_vide par la valeur M
Sem_Init (Decollage_plein, 0); Sem_Init (Atterrissage_vide, N); Sem_Init (Atterrissage_plein, 0);
Sem_Init (Piste, 1);
```

```
void SortirAvions ( ) (
{
Down(Decollage_vide) ;
Ajouter_un_avion_dans_la_zone_attente_decollage( );
Up(Decollage_plein) ;
}

void Decollage ( )
{
Down(Decollage_plein) ;
Down(Piste) ;
Faire_decoller_un_avion ( );
Up(Piste) ;
Up(Decollage_vide) ;
}

void Atterrissage ( )
{
Down(Atterrissage_plein) ;
Down(Piste) ;

Faire_Atterrir_un_avion ( );

Up(Piste) ;
Up(Atterrissage_vide) ;
}

void AmenerAvions ( )
{
Down(Atterrissage_vide) ;
Ajouter_un_avion_dans_la_zone_attente_atterrissage( );
Up(Atterrissage_plein) ;
}
```

Exercice 3 : Unité de fabrication des stylos

Synchronisez au moyen de sémaphores l'enchaînement des opérations de fabrication de stylos à bille. Chaque stylo est formé d'un corps, d'une cartouche, d'un bouchon arrière et d'un capuchon. Les opérations à effectuer sont les suivantes :

1. remplissage de la cartouche avec l'encre (opération RC),
2. assemblage du bouchon arrière et du corps (opération BO),
3. assemblage de la cartouche avec le corps et le capuchon (opération AS),
4. emballage (opération EM).

Chaque opération est effectuée par une machine spécialisée (mRC, mBO, mAS, mEM). Les stocks de pièces détachées et d'encre sont supposés disponibles quand la machine est disponible.

- Les opérations RC et BO se font en parallèle.

- L'opération AS doit être effectuée, après ces deux opérations, en prélevant directement les éléments sur les machines mRC et mBO.
- Le produit assemblé est déposé dans un stock en attente de l'opération EM.
- L'opération EM se fait donc après AS, à partir du stock. Le stock est supposé de taille N et de discipline FIFO

Semaphore SRC=1, SBO=1, SAS1=0, SAS2=0, libre=N, occupe=0;

mRC( )
{
while (1) {
P(SRC) ;
RC( ) ;
V(SAS1) ;
}
}
	mBO( )
	{
	while (1)
	{
	P(SBO) ;
	BO( ) ;
	V(SAS2) ;
	}
	}
		mAS( ) {
		while (1)
		{
		P(SAS1) ;
		P(SAS2) ;
		P(libre) ;
		AS( ) ;
		V(SRC) ;
		V(SBO);
		V(occupe) ;
		}
		}
			mEM( )
			{
			while (1)
			{
			P(occupe) ;
			EM() ;
			V(libre) ;
			}
			}

Exercice 4 :

Deux villes A et B sont reliées par une seule voie de chemin de fer. Les trains peuvent circuler dans le même sens de A vers B ou de B vers A. Mais, ils ne peuvent pas circuler simultanément dans les sens opposés. On considère deux classes de processus: le nombre N de trains allant de A vers B (Train Avers B) et le nombre M de trains allant de B vers A (Train B versA). Ces processus se décrivent comme suit :

Train AversB :

1. Demande d'accès à la voie par A ;
2. Circulation sur la voie de A vers B;
3. Sortie de la voie par B;

Train BversA :

1. Demande d'accès à la voie par B ;
2. Circulation sur la voie de B vers A;
3. Sortie de la voie par A;

L'exercice représente le Modèle des lecteurs et des rédacteurs (la voie joue le rôle de la base de données).

2) Sémaphore autorisation =1 ; // Le sémaphore autorisation est partagé par tous les trains

Sémaphore mutex =1 ; //Le sémaphore mutex est partagé par tous les trains allant dans le même sens. // Il y a donc deux sémaphores mutex un pour chaque sens.

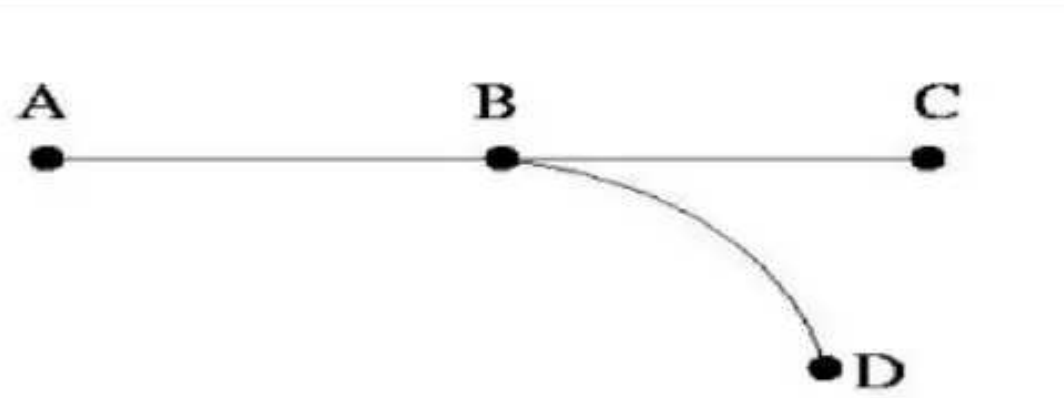
Demande d'accès par un train AversB	Sortie de la voie par B
P(mutex) Si NbAB ==0 alors P(autorisation) ; NbAB=NbAB+1 ; V(mutex) ;	P(mutex) Si NbAB ==1 alors V(autorisation) ; NbAB=NbAB-1 ; V(mutex) ;

Demande d'accès par un train BversA	Sortie de la voie par A
P(mutex) Si NbBA ==0 alors P(autorisation) ; NbBA=NbBA+1 ; V(mutex) ;	P(mutex) Si NbBA ==1 alors V(autorisation) ; NbBA=NbBA-1 ; V(mutex) ;

Exercice 5 :

On dispose d'une carte électronique à base de microcontrôleurs pour contrôler un ensemble de robots. La carte est livrée avec un logiciel sous Linux, qui permet de créer son propre programme pour commander et coordonner un ensemble de robots, et de le charger ensuite dans la mémoire non volatile (sur la carte) par un port série.

On vous sollicite pour écrire un pseudocode qui contrôle le déplacement de plusieurs robots sur les chemins suivants :



Les robots peuvent partir de A vers C ou de D vers A

Pour éviter tout risque de collision, il faut s'assurer que chaque segment du chemin (segments AB, BC et DB) est utilisé par un robot au plus.

Pour répondre à cette question, complétez le pseudocode suivant afin que les règles ci- dessus soient respectées.

Dans ce programme, chaque robot est commandé par un processus.

```
 /*0*/
 void TraverserSegAB ( ) ; // Traverser le segment AB
void TraverserSegBC ( ) ; // Traverser le segment BC
void TraverserSegBD ( ) ; // Traverser le segment BD
Processus RobotAC {
 /* 1 */
TraverserSegAB ()

/*2*/
TraverserSegBC( ) ;
  /*3*/
}
Processus  RobotDA ()
{
/*4*/
TraverserSegBD()
/*5*/
TraverserSegAB( );
 /* 6*/
}
```

```
/*0*/ Semaphore SAB =1, SBC=1, SBD=1 ;
void TraverserSegAB ( ) ; // Traverser le segment AB
void TraverserSegBC ( ) ; // Traverser le segment BC
void TraverserSegBD ( ) ; // Traverser le segment BD
```

Processus RobotAC	Processus RobotDA

{
/* 1 */ P(SAB) ;
TraverserSegAB ( ) ;
/* 2*/ V(SAB) ; P(SBC) ;
TraverserSegBC( ) ;
/*3*/ V(SBC) ;
}
	{
	/* 4 */ P(SBD) ;
	TraverserSegBD( ) ;
	/*5*/ P(SAB) ; V(SBD) ;
	TraverserSegAB( );
	/* 6*/ V(SAB)
	}

Exercice 6 : Synchronisation à l'aide de sémaphores

La capacité maximale d'un lave-auto est de 1 camion ou 3 voitures (c.-à-d. on peut avoir jusqu'à 3 voitures au lavage au même temps, par contre on ne peut pas avoir plus d'un camion à la fois). On vous demande de gérer l'accès au lave-auto de sorte que la capacité maximale du lave-auto soit respectée. Le lave-auto a une seule entrée et une seule sortie.

Supposez que les camions et les voitures sont des threads qui demandent périodiquement l'accès au lave-auto.

Question

Supposant l'existence de deux types de processus : processus camion qui contient deux fonctions : acces_camion() et sortie_camion() et processus voiture qui contient deux fonctions acces_voiture() et sortie_voiture().

Reproduisez les fonctions ci-dessous sur votre copie et proposer une solution en synchronisant l'accès au lave- auto en utilisant les sémaphores. Votre solution ne doit privilégier ni les camions, ni les voitures.

```
Semaphore tour =....., acces = .....;
// tour : sémaphore pour traiter les voitures et les camions selon leur ordre d'arrivée
// acces : sémaphore pour gérer les accès au lave-auto
```

void acces_voiture()	Void sortie_voiture()	Void acces_camion()	void sortie_camion()
{	{	{	{
P(.....);	V(.....);	P(.....);	V(.....);
P(.....);	}	P(.....);	V(.....);
V(.....);		P(.....);	V(.....);
		P(.....);	}
}		V(.....);	
		}	

Semaphore tour =1, acces = 3;

// tour : sémaphore pour traiter les voitures et les camions selon leur ordre d'arrivée // acces : sémaphore pour gérer les accès au lave-auto

void acces_voiture()	Void sortie_voiture()	Void acces_camion()	void sortie_camion()
{	{	{	{
P(acces)	V(acces)	P(acces)	V(acces)
P(tour)	}	P(acces)	V(acces)
V(tour)		P(acces)	V(acces)
}		P(tour)	
		V(tour)	}
		}	

Exercice 7 « Sémaphores » :

Une piscine peut contenir ***N*** nageurs au ***maximum***. Le nombre ***N*** représente le nombre des ***paniers*** disponibles pour les habits des nageurs. A l'entrée comme à la sortie les nageurs entrent en concurrence pour l'obtention d'une cabine d'habillage/déshabillage. Nous supposons qu'il y a ***C cabines*** tel que ***1 <= C << N***. On peut assimiler ces nageurs à des processus concurrents. Chaque processus effectue les opérations suivantes :

```
Processus_Nageur
Début
< se déshabiller >
< nager >
< se rhabiller >
Fin
```

Question :

1. En reproduisant la fonction ci-dessous sur votre copie, proposez à base des sémaphores, une solution de synchronisation d'accès aux ressources partagées *(les paniers et les cabines)* entre les nageurs.

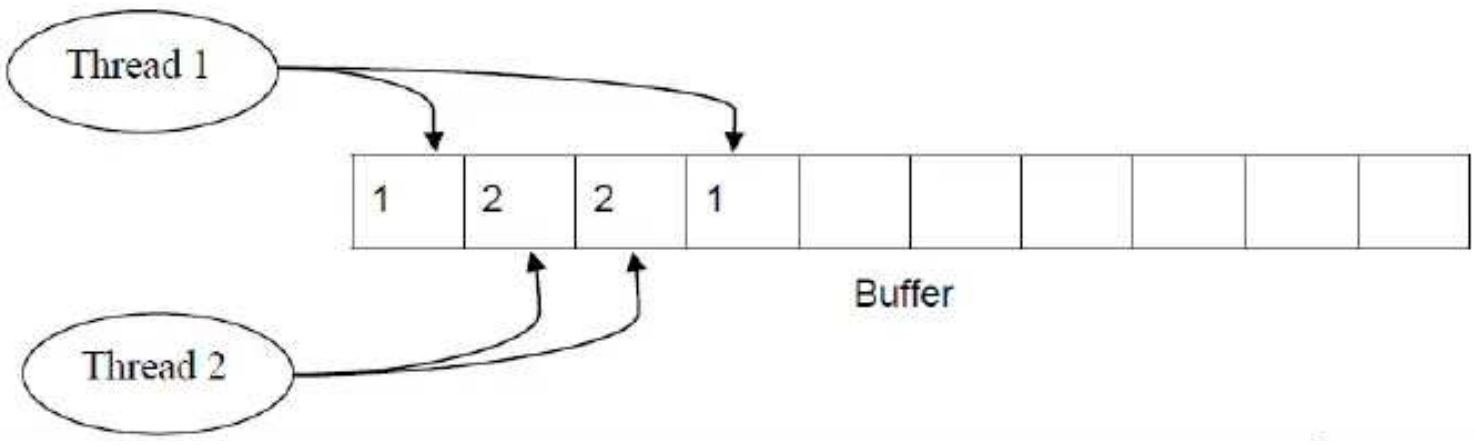
```
var S1,S2 : semaphore init ……,…… // semaphore à initialiser
Processus Nageur
```

```
Début
……
……
<se déshabiller>
……
<nager>
……
<se rhabiller>
……
……
Fin
```

```
var S1,S2 : semaphore init C,N // semaphore à initialiser
//les deux sémaphores sont créés pour gérer les paniers (S1) et les cabines (S2)
Processus Nageur
    Début
    P(S1)
     P(S2)
     <se déshabiller>
    V(S2)
    <nager>
     P(S2)
    <se rhabiller>
    V(S2)
    V(S1)
Fin
```

Exercice 8 Threads :

Ecrire un programme c sous Linux permettant de lancer deux threads. Les deux threads partagent un buffer d'entiers de taille limitée N. Le premier thread écrit, tant qu'il y en a de place vide, l'entier « 1 ». Le second thread écrit l'entier « 2 ».



Compléter les blocs manquants du programme suivant :

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define N 1000
volatile int tab[N];
volatile int CompteurPremiereCaseVide=0;
/* 1 */

void *thread1Fonc (void * arg)
{
      while(1)
       {
/* 2 */
            if(CompteurPremiereCaseVide<N)
            {
                  /* 3 */

            }else
            {
                 /* 4 */

            }
       }
}
void *thread2Fonc (void * arg)
{
                /* 5 */
}
int main ()
{
  pthread_t th1, th2;

                /* 6 */
 if (pthread_create (&th1, NULL, thread1Fonc, NULL) < 0)
      {
      perror("pthread_create erreur>>Thread 1:");
      exit (EXIT_FAILURE);
      }
            /* 7 */
           /* 8 */
}
```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define N 5

volatile int tab[N];

volatile int CompteurPremiereCaseVide=0;

/* 1 */

```
static pthread_mutex_t my_mutex1; //case du tableaux
static pthread_mutex_t my_mutex2; //compteur

void *thread1Fonc (void * arg)

{

    while(1)

    {
/* 2 */
pthread_mutex_lock(&my_mutex2);
        if(CompteurPremiereCaseVide<N)

        {

            /* 3 */
   pthread_mutex_lock(&my_mutex1);
   tab[CompteurPremiereCaseVide]=1;
   CompteurPremiereCaseVide++;
   pthread_mutex_unlock(&my_mutex1);
   pthread_mutex_unlock(&my_mutex2);
 }else

        {

            /* 4 */
   pthread_mutex_unlock(&my_mutex2);


        }

    }
pthread_exit(0);

}

void *thread2Fonc (void * arg)

{

    while(1)

    {
/* 2 */
```

```
pthread_mutex_lock(&my_mutex2);
        if(CompteurPremiereCaseVide<N)

        {

            /* 3 */

   pthread_mutex_lock(&my_mutex1);
   tab[CompteurPremiereCaseVide]=2;
   CompteurPremiereCaseVide++;
   pthread_mutex_unlock(&my_mutex1);
   pthread_mutex_unlock(&my_mutex2);
 }else

        {

           /* 4 */
   pthread_mutex_unlock(&my_mutex2);


        }
    }
pthread_exit(0);
}

int main ()

{
int i;

  pthread_t th1, th2;



          /* 6 */
pthread_mutex_init(&my_mutex1, NULL);
pthread_mutex_init(&my_mutex2, NULL);

 if (pthread_create (&th1, NULL, thread1Fonc, NULL) < 0)

    {

    perror("pthread_create erreur>>Thread 1:");
```

```
    exit (EXIT_FAILURE);

    }
        /* 7 */
    if (pthread_create (&th2, NULL, thread2Fonc, NULL) < 0)


    {
    perror("pthread_create erreur>>Thread 1:");

    exit (EXIT_FAILURE);

    }

       /* 8 */
    (void)pthread_join(th1, NULL);
    (void)pthread_join(th2, NULL);

}
```