Partie 4 : Synchronisation
Le corrigé

Solution 1

a. Les sémaphores :

Mutex0 et mutex1 pour contrôler les accès aux tampons.

Vide1, Vide0, Plein1 et Plein0 pour bloquer un processus si le tampon est vide ou plein.

b. Semaphore

mutex1=1, mutex0=1, Vide0=N, Vide1=N, Plein0 = 0, Plein1 = 0;

```
P0
Message m, mc;
int ip=0;
Répéter
{       m= lire();
        mc = Encrypter(m);
        P(Vide0);
        P(mutex0);
        T0[ip] = mc;
        V(mutex0);
        ip = ip+1 mod(N)
        V(Plein0);
}
```

```
P1
int icp=0;
Répéter
{       P(Plein0);
        P(Vide1)
        P(mutex0);
        P(mutex1)
        T1[icp] = T0[icp];
        V(mutex1);
        V(mutex0)
        icp = icp+1 mod(N)
        V(Plein1);
        V(Vide0);
}
```

```
P2
Message mc;
int ic=0;
Répéter
{       P(Plein1)
        P(mutex1);
        mc = T1[ic];
        V(mutex1);
        ic = ic+1 mod(N)
        V(Vide1);
        Envoyer(mc);
}
```

Solution 2

char T[N]; // tableau de N caractères
Semaphore Plein =0, Vide=N, Mutex=1

```
Producteur {
 int ip=0, M;
 char ch[N];
  Repeter
  {
        M=Lire(ch,N);
        Pour i=1 à M Pas 1 faire P(Vide);
        P(Mutex);
        Deposer(ch, M, ip);
        V(mutex);
        Pour i=1 à M Pas 1 faire V(Plein);
        ip = (ip + M) % N;
  }
}
```

```
Consommateur {
 int ic=0;
 char c;
 Repeter
  {
        P(Plein);
        P(Mutex);
        c = Retirer( ic);
        V(Mutex);
        V(Vide);
        ic = (ic+1) %N ;
        Traiter(c);

   }
}
```

1. Le risque de famine

2. Le moniteur regroupe, en une seule structure, toutes les sections critiques d'un problème donné. Le contrôle d'accès aux sections critiques est géré par le moniteur. Il facilite ainsi la compréhension et l'implémentation du code de synchronisation.

Solution 3

1.
Un seul peut manger à la fois. Un sémaphore binaire mutex suffit :
Semaphore mutex =1 ;
Fonction Shaolin(i in [0,2])

```
{
        P(mutex) ;
        PrendreFourchettes (i);
        Manger() ;
        ReposerFourchettes (i) ;
        V(mutex) ;
}
```

2.

```
#define NB_BUFF 2
#define BUFF_SIZE 4
sem_t sem_libre[NB_BUFF];
sem_t sem_occupe[NB_BUFF];
int tab[NB_BUFF][BUFF_SIZE];

void *ordonnanceur(void *inutilise)
{
        //buff numéro du buffer courant (buff < NB_BUFF)
        // i[buff] indice dans le buffer courant. (i[buff]<BUFF_SIZE, buff =1,NB_BUFF)
        int i[NB_BUFF] , buff ;

        for(buff :=0 ; buff<NB_BUFF ; buff++) i[buff]=0 ;
        buff=0 ;
        while (1)
        { /*0*/
                if ((sem_trywait(&sem_libre[buff]) == -1) ) {
                        buff = (++buff)% NB_BUFF;
                        continue;
                }
                tab[buff][i[buff]++%BUFF_SIZE] = prochainProc();
                sem_post(&sem_occupe[buff]);
        }
}
```

Solution 4

1. Semaphore S1=1, S2=1, S13=0, S23=0 ;

```
P1 ()
{
   int n=0;
   while(true)
   {
      P(S1);
       printf("cycle %d de
        %d", n++, i);
         V(S13) ;
   }
}
```

```
P2 ()
{
   int n=0;
   while(true)
   {     P(S2);
         printf("cycle  %d  de
         %d", n++, i);
         V(S23) ;
   }
}
```

```
P3 ()
{   int n=0;
    while(true)
  {  P(S13);
     P(S23);
      printf("cycle %d de
       %d", n++, i);
   V(S1) ;
   V(S2) ;
  }
}
```

2. 10 producteurs, 10 consommateurs avec une liste infinie

1 séquencement.

1 thread de syncrhonisation des feux

3 sémaphores de séquencement :

1 pour aller tout droit ou tourner a droite dans le sens est-ouest (init : 1)

1 pour aller tout droit ou tourner a droite dans le sens nord-sud (init : 0)

1 pour tourner a gauche dans le sens nord-sud (init : 0)

1 thread producteur par voies de départ : 10 threads producteurs

1 liste par voie productrice : 10 listes

1 semaphore de productions par liste : 10 semaphores de production (initialisé a 0)

1 thread consommateur par voies d'arriver : 10 threads consommateurs

1 mutex par liste : 10 mutex (parce que les listes sont partagées entre les prod/cons)

Solution 5

1.

```
semaphore  SA=0, SB=0 ;  /*0*/
char T[256] ;
void depot (char buf[] ) ;
void recuperer(char buf[] ) ;
```

Processus A

```
{  char mess[256] , rep[256];

        while (1)
        {       /* 1 */
              lire (mess);
              depot (mess ) ;


              /* 2*/ V(SB);
                     P(SA);
                recuperer(rep) ;
              /*3*/
        }

}
```

Processus B

```
{    char mess[256], rep[256]  ;

 while (1)
 {       /*  4 */
          P(SB);
         recuperer( mess) ;
          reponse(mess,rep);


          /*5*/
          depot(mess);
          /* 6*/ V(SA);
 }

}
```

2.

```
semaphore  SA=0, SB=0, SC=0, mutex=1 ;  /*0*/
char T [257] ;
void depot (char buf[] ) ;
void recuperer(char buf[] ) ;
```

Processus A

```
{
   char mess[257], rep[256];

   while (1)
   {          P(mutex);
              lire (mess); mess[256]='A',
              depot (mess ) ;

              V(SB);
              P(SA);
              recuperer(rep) ;
             V(mutex) ;
   }

}
```

Processus B

```
{
    char mess[257],rep[256];

  while (1)
  {
     P(SB);
      recuperer( mess) ;
      reponse(mess,rep);
      depot(rep);
      if (mess[256]='A') V(SA);
     else V(SC);

  }

 }
```

Processus C

```
   {
    char mess [257], rep[256] ;

        while (1)
        {    P(mutex);
             lire (mess); mess[256]='C',
             depot (mess ) ;
             V(SB);
             P(SC);
             recuperer(rep) ;
             V(mutex);
        }

    }
```

Solution 6

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) ;
 }
}
```

Solution 7

1)

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

2) Toutes les demandes d'accès sont mémorisées dans les files des sémaphores. Si les files sont gérées FIFO et la durée de traversée de chaque segment est fini alors pas de problème de famine. Dans le cas contraire, il pourrait y avoir famine.
Pas d'interblocage entre les robots car :

- Si un robot RobotAC attend AB, il ne détient aucun autre segment (pas de détention et attente).
- Si un robot RobotAC détient AB et attend BC qui est alloué à un autre robot, ce dernier n'est pas en attente d'un autre segment (pas de détention et attente).
- Si un robot RobotDA attend BD, il ne détient aucun autre segment et donc ne bloque pas les autres (pas de détention et attente).
- Si un robot RobotDA détient BD et attend AB qui est alloué à un autre robot (robotDA ou robotAC), ce dernier ne se mettra pas en attente de BD (pas d'attente circulaire).

Solution 8

1) Sémaphores utilisés :

Semaphore SI1=1, SI2=1, SF1=1, SF2=0;

```
Changement
{
        int Feu = 1
        while (1)
        {       sleep (Duree_du_feu) ;
                if (Feu == 1)
                {       P(SF1);
                        V(SF2);
                        Feu = 2;
                }
                else
                {       P(SF2);
                        V(SF1);
                        Feu = 1;
                }
        }
}
```

```
Traversee1
{
        P(SI1);
        P(SF1);
        Traversee() ;
        V(SF1);
        V(SI1);
}
```

```
Traversee2
{
        P(SI2);
        P(SF2);
        Traversee() ;
        V(SF2);
        V(SI2);
}
```

SI1 et SI2 sont introduits pour éviter que les voitures attendent sur SF1 et SF2 et bloquent de ce fait les changements de feu effectués par Changement.

2) Moniteurs :

```
Moniteur Intersection
{       Boolc wtour1, wtour2, wfeu1, wfeu2 ;
        int Feu = 1, nbw1 = 0, nbw2 = 0;

        Changement()
        {       while (1)
                {       sleep (Duree_du_feu) ;
                        if (Feu == 1)
                        {       Feu = 2;
                                signal (wfeu2) ;
                        }
                        else
                        {       Feu = 1;
                                signal (wfeu1) ;
                        }
                }
        }

Traversee1()
{
        if( nbw1 != 0)
        {       nbw1++ ;
                wait(wtour1) ;
        }
        if( Feu != 1)
            wait(wfeu1) ;
        circuler() ;
        if( nbw1 != 0)
        {       nbw1-- ;
                signal(wtour1);
        }
}

{
        if( nbw2 != 0)
        {       nbw2++ ;
                wait(wtour2) ;
        }
        if( Feu != 2)
             wait(wfeu2) ;
        circuler() ;
        if( nbw2 != 0)
        {        nbw2-- ;
                signal(wtour2);
        }
 }

} // fin du moniteur
```

Traversee2()

Solution 9

```
struct CompteurEvenement
{
      int val ;
      /* 0 */  semaphore mutexE ;
      struct listeSem {semaphore  psem ; int  pval ;} * Lsem ;
}

void Await (CompteurEvenement *E, int Valeur)
{        P( E->mutexE) ;
         if (E->val < Valeur}
         {  /*1*/ semaphore *  nsem = new (semaphore) ;
                  initialiser le sémaphore pointé par nsem à  0 ;
                   ajouter  (*nsem, Valeur)  à E->Lsem;
                  V(E->mutexE);
                  P(*nsem);
         } else V(E->mutexE) ;
}

void Advance (CompteurEvenement *E)
{
        /* 2 */    P( E->mutexE) ;
                 E->val = E->val + 1 ;
                 Pour chaque élément (sem, v) de E->Lsem  telle que E->val <= v
                 {   V(sem) ;  supprimer  cet élément ; }
                 V(E->mutexE) ;
    }

int Read (CompteurEvenement *E)
{    /* 3 * /
         int v;
         P( E->mutexE) ;
         v = E->val ;
         V( E->mutexE) ;
         return v;
    }

 CompteurEvenement CE = (0, 1, NULL) ;
```

Solution 10

```
Semaphore quitte=0 , passage=0 ; present=0; /*0*/
Contrôleur ()                                   Train()
{                                               {
  while(1)                                              V(present) ;  /*3*/
  {       P(present)      /*1*/                         P(passage);
          FermerBarrieres() ;                           Traverser();
          V(passage) ; /*2*/                            V(quitte) ; /*4*/
          P(quitte) ;                           }
          While (PNB(present))
          {       V(passage) ;
                  P(quitte)} ;
                  OuvrirBarrieres() ;
          }
}
```

//present est le sémaphore qui sert à comptabiliser le nombre de trains présents.

//passage est le sémaphore qui mémorise les trains en attente du passage à niveau.

//quitte est le sémaphore qui sert à bloquer/débloquer le contrôleur lorsque le train entame/termine la traversée.

// PNB est l'équivalent de sem_trywait.

Solution 11

Cette implémentation est incorrecte pour plusieurs raisons :

- on désinscrit toujours l'utilisateur de son premier cours, même s'il n'a pas pu être inscrit au deuxième cours ;

- du fait que l'on verrouille le deuxième cours alors qu'on a déjà un verrou sur le premier, on a un risque d'interblocage si deux étudiants lancent simultanément la routine avec les deux mêmes valeurs de cours, mais dans l'ordre inverse : chacun verrouillera d'abord le cours que l'autre voudra verrouiller ensuite, et il ne sera donc pas possible de sortir de cet interblocage ;

- on ne verrouille pas le deuxième cours avant de faire le test pour savoir s'il est plein.

```
void EchangeCours (Putilisateurs utilisateur, PCours cours1, cours2) {
        cours2->verrouille ();
        if (cours2->estPlein == false) {
                cours2->inscrit (utilisateur);
                cours2->deverrouille ();
```

```
                cours1->verrouille ();
                cours1->desinscrit (utilisateur);
                cours1->deverrouille ();
        }
        else
                cours2->deverrouille ();
}
```

Solution 12

a)

```
class Barrier_t
{       int nbproc ;
         int cp ;
        semaphore mutex , sem_barriere ;
        public : Barrier_t (int) ;
                void Barrier ( ) ;
}
Barrier_t ::Barrier_t (int v)
{   nbproc = v ; cp=0 ; mutex = 1 ; sem_barriere=0 ;}

void  Barrier_t ::Barrier()
{       P(mutex);
        cp++ ;
        if (cp==nbproc)
        {       for (int i=0 ; i<cp-1; i++)
                                V(sem_barriere);
                cp=0;
                V(mutex);
        } else {
                  V(mutex);
                  P(sem_barriere);
                }
}
```

b) Barrier_t /*0*/ E1(3), E2(3), E3(2), E4(2);

```
mRC( )
{  while (1)
   {       RC( ) ;
           E1.Barriere();
           E2.Barriere();
   }
}

mBO( )
{   while (1)
   {      BO( ) ;
           E1.Barriere();
           E2.Barriere();
    }
}

mAS( )
{    while (1)
   {       E1.Barriere() ;
           GP() ;
           E2.Barriere() ;
           AS( ) ;
           E3.Barriere() ;
           E4.Barriere();
    }
}

mEM( )
{   while (1)
   {       E3.Barriere() ;
           GP();
           E4.Barriere();
           EM( ) ;
    }
}
```

c) Oui, par exemple, deux processus A et B partagent deux barrières E1 et E2 ;

Processus A {E1.Barriere() ; E2.Barriere(); ...}

Processus B { E2.Barriere(); … E1.Barriere(); }

Les deux processus se bloquent mutuellement.

Solution 13

1) Non, car si P2 est en section critique et P1 a exécuté P(mutex1) alors P1 est bloqué et empêche P3 d'entrer en section critique.

Condition non vérifiée : Un processus en dehors de sa section critique bloque un autre processus.

Processus P1

```
P(mutex1) ;
n=n-1 ;
V(mutex1) ;
P(mutex2) ;
Out = out +1 ;
V(mutex2) ;
```

2) On va utiliser un vecteur T de n booléens. T[i] est 0 si le calcul de la ligne i n'est pas encore entamé. T[i] est égal à 1 sinon. Ici, on n'a pas besoin de sémaphores pour les accès aux matrices (en lecture uniquement). Par contre, on a besoin d'un sémaphore binaire mutex pour contrôler les accès au vecteur T. Initialement, tous les éléments de T sont nuls.

```
fonction CalculLignes ( ){
   pour i = 0 à n-1 pas 1
   { P(mutex)
        si ( T[i]==0)
        {       T[i] = 1 ;
                V(mutex) ;
                Pour j = 1 à n pas 1
                {       Pour k=1 à n pas 1
                                R[i,j] += A[i,k] * B[k,j] ;
                }
        } sinon V(mutex) ;
     }
}
```

Solution 14

1) 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

P(mutex)

Si NbAB ==0 alors P(autorisation) ;

NbAB=NbAB+1 ;

V(mutex) ;

Sortie de la voie par B

P(mutex)

Si NbAB ==1 alors V(autorisation) ;

NbAB=NbAB-1 ;

V(mutex) ;

Demande d'accès par un train BversA

P(mutex)

Si NbBA ==0 alors P(autorisation) ;

NbBA=NbBA+1 ;

V(mutex) ; Sortie de la voie par A

P(mutex)

Si NbBA ==1 alors V(autorisation) ;

NbBA=NbBA-1 ;

V(mutex) ;

Solution 15

a) Semaphore Mutex =1, Vide=Max, Plein=0 ;

Message tampon [Max] ;

int ip =0 ; // ip devient global

int ic =0 ; // ic devient global

```
Producteur ( int i)
{           Message m ;
            Repeter
            {       m = creermessage( ) ;
                    P(Vide) ;
                    P(Mutex) ;
                    Tampon[ip]=m;
                    ip++ ; // ip++ est dans la section critique
                    V(Mutex) ;
                    V(Plein) ;
            }tant que vrai ;
}
Consommateur( int i )
{           Message m ;
            Repeter
            {       P(Plein) ;
                    P(Mutex) ;
                    m = Tampon[ic]; ic++ ; // ic est partagé entre tous les consommateurs
                    V(Mutex) ;
                    V(Vide) ;
            }tant que vrai ;
}
```

b) Semaphore

Mutex [n] = {1, 1, ….,1},

Vide [n] = {Max, Max, …, Max},

```
Plein [n] = {0, 0, ., 0} ;
Message tampon [n][Max] ;
Producteur ( )
{               int ip =0 ; // un seul producteur
                Message m ;
                Repeter
                {       m = creermessage() ;
                        pour i=0, n- 1, pas 1
                        faire
                        P(Vide[i]) ;
                        P(Mutex[i]) ;
                        Tampon[i][ip]=m;
                        V(Mutex[i]) ;
                        ip++ ;
                        V(Plein[i]) ;
                }tant que vrai ;
}
Consommateur( int i)
{               int ic =0 ;
                Message m ;
                Repeter
                {       P(Plein[i]) ;
                        P(Mutex[i]) ; m = Tampon[i][ic];
                        V(Mutex[i]) ;
                        ic++ ;
                        V(Vide[i]) ;
                }tant que vrai ;
}
```