Université Toulouse III – Paul Sabatier
118 route de Narbonne
31062 Toulouse cedex 9

Éléments de correction des TD

Remarque : Les solutions sont écrites ici dans un « pseudo-code ». Pour programmer ces applications en C ou Java, il faut utiliser les IPC Posix ou les threads Java et classes de synchronisation associées (ce qui est le but des TP).

TD1 – Réseaux de Petri

Voir cours et exercices supplémentaires

TD2 – Sémaphores – Affichage alterné

Globalement, un seul accès à l'écran est possible en même temps, il faut donc assurer une exclusion mutuelle entre les deux processus. De plus, un processus qui vient d'afficher doit libérer l'écran pour l'autre processus et ne pas pouvoir récupérer cet accès pour lui-même.

```
Sémaphore Mutex1, Mutex2 ;
Init(Mutex1, 1) ;  /* On choisit arbitrairement le 1er processus qui affiche */
Init(Mutex2, 0) ;
void Processus1 () {
   while (1) {
      . . .
      /* Demander à accéder à l'écran */
      P(Mutex1) ;
      /* Afficher . . . */
      /* Libérer l'accès pour l'autre */
      V(Mutex2) ;
      . . .
   }
}
void Processus2 () {
   while (1) {
      . . .
      /* Demander à accéder à l'écran */
      P(Mutex2) ;
      /* Afficher . . . */
      /* Libérer l'accès pour l'autre */
      V(Mutex1) ;
      . . .
   }
}
```

TD2 – Sémaphores – Généralisation à N processus de l'affichage alterné

Utiliser N sémaphores d'exclusion mutuelle (=> un tableau).

Un seul est initialisé à 1, les autres à 0 (un seul accès global).

Un processus prend un jeton dans le sémaphore correspondant à son rang et le repose dans le sémaphore de rang suivant (module N).

TD2 – Sémaphores – Producteurs-Consommateurs – Variante de base

Un producteur doit déposer son message dans l'ordre croissant (modulo N) des indices du tableau représentant le buffer, en commençant par la première case. Les producteurs vont donc partager l'indice de la prochaine case dans laquelle déposer un message.

On doit donc synchroniser les accès des producteurs à cet indice => sémaphore d'exclusion mutuelle.

Un consommateur fait de même, il retire dans l'ordre des dépôts. Les consommateurs partagent l'indice de la prochaine case depuis laquelle retirer un message.

On doit donc synchroniser les accès des consommateurs à cet indice => sémaphore d'exclusion mutuelle.

Le buffer est partagé par les deux types de processus, on doit synchroniser leurs accès. Un producteur ne peut plus déposer si le buffer est plein. Un consommateur ne peut retirer si le buffer est vide. Il est donc nécessaire de compter le nombre de cases occupées. De plus, un producteur est seul à pouvoir débloquer un consommateur en attente en créant une case pleine. Inversement, un consommateur est seul à pouvoir débloquer un producteur qui attend une case vide.

```
Voir exemple 5 du cours.
```

TD2 – Sémaphores – Producteurs-Consommateurs – Variante 1 : dépôts alternés

La synchronisation entre producteurs et consommateurs ne change pas.

La synchronisation entre producteurs pour l'accès à l'indice de retrait ne change pas.

La synchronisation entre producteurs doit assurer l'alternance des dépôts. Elle est semblable à celle mise en place pour l'alternance des affichages et nécessite deux sémaphores d'exclusion mutuelle. Cette synchronisation assure qu'un seul producteur accède à l'indice partagé, le sémaphore d'exclusion mutuelle entre producteurs devient donc inutile.

```
#define N …
int indPremCasePleine = 0, indPremCaseVide = 0 ;
Message BufferMessages[N] ;   /* Message = structure avec contenu + type */
Sémaphore MutexType[2], MutexC, CasePleine, CaseVide;
Init(MutexType[0], 1) ;       /* Type 1er dépôt choisi arbitrairement */
Init(MutexType[1], 0) ;
Init(MutexC, 1) ;             /* De même pour un consommateur */
Init(CasePleine, 0) ;         /* Aucune case pleine au début
                              => aucun accès pour les consommateurs */
Init(CaseVide, N) ;           /* Toutes les cases sont vides
                              => N accès pour les producteurs */
void Producteur (Message monMsg) {
  while (1) {
    /* Préparer le message à déposer */
    /* Mon message est-il du bon type ? */
    P(MutexType[monMsg.type]) ;
    /* Peut-on avoir une case vide ? */
    P(CaseVide) ;
    /* Déposer le message */
    BufferMessages[indPremCaseVide] = monMsg ;
    indPremCaseVide = (indPremCaseVide + 1)%N ;
    /* On a créé une case pleine pour un consommateur */
    V(CasePleine) ;
    /* Donner l'accès à l'autre type de dépôt */
```

```
      V(MutexType[(monMsg.type + 1) % 2] ;
      . . .
   }
}
```

Le code d'un consommateur est inchangé.

TD2 – Sémaphores – Producteurs-Consommateurs – Variante 2 : retraits demandés

On a besoin de compter les cases pleines (ou vides) pour savoir quand on a le 1er dépôt ou quand il reste des cases pleines ➔ variable partagée : nbCasesPleines, initialisée à 0 ➔ besoin de protéger l'accès à cette variable puisque producteurs et consommateurs peuvent la modifier ➔ sémaphore d'exclusion mutuelle MutexCP, initialisé à 1.

```
void Producteur (Message monMsg) {
   while (1) {
      /* Préparer le message à déposer */
      /* Peut-on avoir une case vide ? */
      P(CaseVide) ;
      /* Déposer le message */
      P(MutexP) ;
      BufferMessages[indPremCaseVide] = monMsg ;
      indPremCaseVide = (indPremCaseVide + 1)%N ;
      /* E.M. sur le nombre de cases pleines */
      P(MutexCP) ;
      nbCasesPleines++ ;
      /* On a créé une 1re case pleine pour un consommateur */
      if (nbCasesPleines == 1) {
         V(CasePleine[monMsg.type]) ;
      }
      V(MutexCP) ;
      V(MutexP) ;
   }
}
void Consommateur (int typeAttendu, Message *monMsg) {
   while (1) {
      /* 1er message à retirer est-il du bon type ? */
      P(CasePleine[typeAttendu]) ;
      /* Retirer le message */
      *monMsg = BufferMessages[indPremCasePleine] ;
      indPremCasePleine = (indPremCasePleine + 1)%N ;
      /* E.M. sur le nombre de cases pleines */
      P(MutexCP) ;
      nbCasesPleines--;
      /* On vient de libérer l'accès à un message d'un certain type, s'il existe,
         et donc l'accès pour un éventuel consommateur */
      if (nbCasesPleines > 0) {
         V(CasePleine[BufferMessages[indPremCasePleine].type]) ;
      }
      V(MutexCP) ;
      /* Et on a créé une case vide pour un producteur */
      V(CaseVide) ;
   }
}
```

TD3 –Sémaphores – Gestion d'une voie unique – Variante 1 : un seul véhicule sur la voie

Ceci se ramène à un problème d'accès en exclusion mutuelle à la voie.

TD3 – Sémaphores – Gestion d'une voie unique – Variante 2 : illimité dans un sens

Cette variante ressemble fortement au problème des lecteurs-rédacteurs. Les véhicules dans le même sens peuvent circuler en « parallèle », les autres sont bloqués. Le premier à accéder à la voie ou le dernier à quitter la voie ont des rôles à part (marquer l'obtention de la voie ou bien libérer effectivement la voie).

```
Sémaphore VU ;          -- Accès à la VU (E.M. sur un sens de circulation)
Sémaphore Mutex[2] ;    -- Un sémaphore d'E.M. par sens de circulation
Init(Mutex[0], 1) ;
Init(Mutex[1], 1) ;
Init(VU, 1) ;
int nbVehiculesEngages[2] ;-- Compteur engagés dans chaque sens, 0 initialement
int sensCourant = 0;       -- Sens de circulation, initialisé arbitrairement

void demanderAcces (int monSens) {
  P(Mutex[monSens]);
  if nbVehiculesEngages[monSens] == 0 {
    P(Vu);
    sensCourant = monSens;
  }
  nbVehiculesEngages[monSens]++;
  V(Mutex[monSens]);
}
void libererAcces (void) {
  /* Nécessite de mémoriser le sens courant qui est donc mon sens
     puisque je sors de la voie */
  int monSens = sensCourant;

  P(Mutex[monSens]);
  nbVehiculesEngages[monSens]--;
  /* Le dernier libère la voie */
  if nbVehiculesEngages[monSens] == 0
    V(Vu);
  /* Un véhicule allant en sens inverse a pu prendre le sémaphore,
  entrer sur la voie unique et changer le sens courant avant que
  je poursuive mon exécution, d'ou la nécessite de la variable
  de sauvegarde monSens à la place de la variable partagée Sens_Courant */
  V(Mutex[Mon_Sens]);
}
```

TD3 – Sémaphores – Gestion d'une voie unique – Variante 3 : limité dans un sens

Puisqu'il s'agit de limiter le nombre de trains engagés à un certain maximum, il suffit donc d'utiliser un sémaphore initialisé à cette valeur maximale.

```
Sémaphore VU ;          -- Accès à la VU
Sémaphore Mutex[2] ;    -- Un sémaphore d'E.M. par sens de circulation
Sémaphore DroitPassage ;      -- Limite le nombre de passages à MAX
Init(Mutex[0], 1) ;
Init(Mutex[1], 1) ;
Init(VU, 1) ;
Init(DroitPassage, MAX) ;
```

```
int nbVehiculesEngages[2] ;  -- Compteur engagés dans chaque sens, 0 initialement
int sensCourant = 0;          -- Sens de circulation, initialisé arbitrairement

void demanderAcces (int monSens) {
  P(Mutex[monSens]);
      if nbVehiculesEngages[monSens] == 0 {
    P(Vu);
    sensCourant = monSens;
  }
      nbVehiculesEngages[monSens]++;
      V(Mutex[monSens]);
  P(DroitPassage) ;
}
void libererAcces (void) {
  int monSens = sensCourant;

      P(Mutex[monSens]);
      nbVehiculesEngages[monSens]--;
  if nbVehiculesEngages[monSens] == 0
         V(Vu);
  V(Mutex[Mon_Sens]);
  P(DroitPassage) ;
}
```

TD4 – Moniteurs de Hoare – Producteurs/consommateurs – Variante 1

```
Variante de base. Voir exemple 2 du cours.
```

TD4 – Moniteurs de Hoare – Producteurs/consommateurs – Variante 2 : Dépôts alternés

1. Conditions de blocage/déblocage :

Un producteur est bloqué si le buffer est plein ou bien si le dernier message déposé est du même type que celui que ce producteur veut déposer. Il est débloqué, par un consommateur, lorsqu'une case devient vide et que cela rend accessible un message du type attendu ou, par un producteur de type opposé lorsque ce dernier le dépose.

Mêmes conditions qu'auparavant pour le consommateur. Un consommateur est bloqué si le buffer est vide. Il est débloqué par un producteur lorsqu'une case devient pleine.

2. Variables d'état / variables conditions :

Il faut savoir où déposer et où retirer le prochain message ➔ variables (partagées) : indCVide, indCPleine

Il faut compter le nombre de cases vides ➔ variable (partagée) du moniteur : nbCasesVides

Il faut connaître le type du prochain message à déposer ➔ variable (partagée) : typeProchainMsg

Il faut bloquer les deux types de producteurs séparément ainsi que les consommateurs ➔ trois variables condition : caseVide[2], casePleine

```
Moniteur Prod_ConsoV2 {

  int nbCasesVides, indCVide, indCPleine, typeDernierMsg ;
  condition caseVide[2], casePleine ;
  const int N = xxx ;
```

```
  typedef struct Message {
    ContenuMessage msg ;
    int type ; } ;
  Message buffer[N] ;
  void depot(Message msg) {
    buffer[indCVide] = msg ;
    indCVide = (indCVide + 1) % N ;
  }
  void retrait(Message *msg) {
    *msg = buffer[indCPleine] ;
    indCPleine = (indCPleine + 1) % N ;
  }
  int oppose(int type) {
    return ((type + 1)%2) ;
  }
  void deposer(Message msg) {
    // Attendre d'avoir la place pour cela et le bon type de
    // message, se bloquer sinon
    if (nbCasesVides == 0 || typeProchainMsg != msg.type)
      caseVide[msg.type].wait() ;
    // Une case vide au moins + bon type → on pourra déposer
    Depot(msg) ;
    // On crée une case pleine
    nbCasesVides-- ;
    // On prépare le prochain dépôt possible
    typeProchainMsg = oppose(msg.type) ;
    // On signale à un éventuel producteur de l'autre type
    // qu'il peut déposer s'il reste une case libre
    if (nbCasesVides > 0)
      caseVide[typeProchainMsg].signal() ;
    // Et on le signale aussi à un consommateur qui attend une case
    casePleine.signal() ;
  }
  void retirer(Message *msg)  {
    // Attendre d'avoir un message à retirer, se bloquer sinon
    if (nbCasesVides != N)
      casePleine.wait() ;
    // Une case pleine au moins → on pourra retirer
    retrait(msg) ;
    // On crée une case vide
    nbCasesVides++ ;
    // On le signale à un producteur du type attendu
    if (nbCasesVides > 0)
      caseVide[typeProchainMsg].signal() ;
   }
  // Initialisation des variables du moniteur
  void init(void) {
    nbCasesVides = N ;
    indCVide = 0 ;
    indCPleine = 0 ;
    typeProchainMsg = 0 ;    // arbitraire
  }
}
```

TD4 – Moniteurs de Hoare – Producteurs/consommateurs – Variante 2 : Choix retrait

1. Conditions de blocage/déblocage :

On n'alterne plus les dépôts donc un producteur est bloqué si le buffer est plein. Il est débloqué, par un consommateur, lorsqu'une case devient vide.

Mais, un consommateur spécifie le type de message qu'il veut retirer donc il est bloqué si le buffer est vide ou si le prochain message à retirer n'est pas du type attendu. Il est débloqué par le premier producteur qui dépose un message (du bon type) ou par un consommateur qui lui permet d'accéder à un message du type attendu en retirant celui se trouvant avant.

2. Variables d'état / variables conditions :

Il faut savoir où déposer et où retirer le prochain message ➔ variables (partagées) : indCVide, indCPleine

Il faut compter le nombre de cases vides (ou pleines) ➔ variable (partagée) du moniteur : nbCasesVides

Il faut connaître le type du premier message à retirer ➔ variable (partagée) : typeProchainRetrait

Il faut bloquer les deux types de consommateurs séparément ainsi que les producteurs ➔ trois variables condition : casesPleines[2], caseVide

TD5 – Moniteurs de Hoare – Lecteurs-Rédacteurs – Variante 1

```
Moniteur LRV1 {
  int nbLecteurs ;
  boolean redacPresent ;
  Condition autorisationLecture, autorisationEcriture ;
  /* Initialisation des variables du moniteur */
   void init(void) {
     nbLecteurs = 0 ;
     redacPresent = false ;
  }
   void debutLire(void) {
      /* La lecture ne pourra pas avoir lieu si un rédacteur est actif */
      if (redacPresent)
         autorisationLecture.wait();
      /* OK On peut lire */
      nbLecteurs++ ;
      /* Réveil en cascade des lecteurs (après un blocage) */
      autorisationLecture.signal();
  }
   void finLire(void) {
     nbLecteurs-- ;
     /* Le dernier lecteur libère un éventuel rédacteur qui attend */
     if (nbLecteurs == 0)
       autorisationEcriture.signal();
  }
  void debutEcrire(void) {
      /* L'écriture ne peut avoir lieu s'il y a déjà un rédacteur
      ou des lecteurs actifs */
      if (redacPresent || nbLecteurs > 0)
         autorisationEcriture.wait();
      redacPresent = true ;
  }
```

```
   void finEcrire(void) {
     redacPresent = false ;
     if (autorisationEcriture.length() > 0)
       /* Réveil d'un rédacteur en priorité si un attend */
       autorisationEcriture.signal() ;
     else
       /* Sinon, réveil d'un lecteur qui réveillera les autres derrière lui */
       autorisationLecture.signal() ;
   }
}
```

TD5 – Moniteurs de Hoare – Lecteurs-Rédacteurs – Variante 2 : priorité rédacteurs

Un rédacteur est prioritaire par rapport aux lecteurs en attente.

```
void debutLire(void) {
  /* La lecture ne pourra pas avoir lieu si un redacteur est actif
    ou si un redacteur EST EN ATTENTE */
  if (redacPresent or autorisationEcriture.length() > 0)
    autorisationLecture.wait();
  /* OK On peut lire */
  nbLecteurs++ ;
  /* Reveil en cascade des lecteurs (apres un blocage) */
  autorisationLecture.wait() ;
}
```

TD5 – Moniteurs de Hoare – Lecteurs-Rédacteurs – Variante 3 : gestion plus équitable

```
   void finEcrire(void) {
     redacPresent = false ;
     if (autorisationLecture.length() > 0)
       /* Réveil d'un lecteur en priorité si un attend */
       autorisationLecture.signal() ;
     else
       /* Sinon, réveil d'un rédacteur */
       autorisationEcriture.signal() ;
   }
```

TD5 – Moniteurs de Hoare – Lecteurs-Rédacteurs – Variante 4 : gestion FIFO

Tous les processus sont bloqués dans une même file d'attente, on a donc une seule variable condition. Certains seront certainement réveillés à mauvais escient et il faut les rebloquer sans favoriser ceux qui étaient bloqués derrière (gestion FIFO). Pour cela, on les rebloque en tête de la file d'attente en utilisant le fait de pouvoir bloquer un processus avec une certaine priorité dans une file d'attente associée à une variable condition : 0 est la priorité la plus forte.

```
Moniteur LRV4 {
  int nbLecteurs ;
  boolean redacPresent ;
  Condition autorisationOperation ;
  /* Initialisation des variables du moniteur */
   void init(void) {
     nbLecteurs = 0 ;
     redacPresent = false ;
  }
   void debutLire(void) {
      /* La lecture ne pourra avoir lieu si un rédacteur est actif
```

```
         ou bien si des processus attendent déjà pour être servis */
      if (redacPresent || autorisationOperation.length() > 0)
         autorisationOperation.wait(1);  /* priorité de 1 pour les nouveaux */
      /* OK On peut lire */
      nbLecteurs++ ;
      /* Il faut réveiller en cascade les lecteurs bloqués derrière moi.
        Mais risque de réveiller un rédacteur */
      autorisationOperation.signal();
  }
   void finLire(void) {
     nbLecteurs-- ;
     /* le dernier libère un éventuel rédacteur qui attend */
     if (nbLecteurs == 0)
       autorisationOperation.signal();
  }
  void debutEcrire(void) {
      /* L'écriture ne peut avoir lieu s'il y a déjà un rédacteur
      ou des processus qui sont déjà en attente d'un accès */
     if (redacPresent || autorisationOperation.length() > 0)
         autorisationOperation.wait(1); /* priorité de 1 pour les nouveaux */
     /* Si j'ai été réveillé, il me faut vérifier qu'aucun lecteur n'est actif */
     if (nbLecteurs > 0)
       autorisationOperation.wait(0);    /* On se rebloque en tête */
     redacPresent = true ;
  }
   void finEcrire(void) {
     redacPresent = false ;
     /* Reveil du suivant bloqué dans la file */
       autorisationOperation.signal() ;
  }
}
```

Remarque

La condition de synchronisation du rédacteur peut être exprimée différemment, mais, quelle que soit la solution, un rédacteur qui arrive ne peut pas passer si :

- il n'est pas le premier de la file et il doit alors attendre son tour ;
- il est le premier et il ne peut pas passer si la ressource est occupée par un autre rédacteur ou par un ou plusieurs lecteurs.

TD6 – Moniteurs de Hoare – Traitement de commandes

```
   Nombre_D_Etapes: constant Natural := 10;
   subtype Numero_D_Etape is Natural range 1..Nombre_D_Etapes;
   Nombre_De_Guichets: constant Natural := 5;
   subtype Numero_De_Guichet is Natural range 1..Nombre_De_Guichets;
   type Type_De_Requete is new Integer;

package body Moniteur_De_Gestion_De_Requetes is

   type Etat_De_Guichet is (Libre, En_Cours, En_Attente);
   type Descripteur_De_Guichet is
   record
      Etat :             Etat_De_Guichet;
```

```
      Etape :            Numero_D_Etape;
      Requete :          Type_De_Requete;
      Fin_De_Traitement: Condition;
   end record;
   Le_Guichet : array (Numero_De_Guichet) of Descripteur_De_Guichet;
   Nombre_De_Guichets_Libres : Natural := Nombre_De_Guichets;
   Un_Guichet_Libre          : Condition;
   Un_Employe : array (Numero_D_Etape) of Condition;

   function Un_Numero_De_Guichet_Libre return Natural is
   begin
      for K in Numero_De_Guichet loop
         if (Le_Guichet(K).Etat = Libre) then
            return K;
         end if;
      end loop;
      return 0;
   end Un_Numero_De_Guichet_Libre;

   function Un_Numero_De_Guichet_Pret_Pour_L_Etape (Ne : in Numero_D_Etape)
      return Natural is
   begin
      for K in Numero_De_Guichet loop
         if (Le_Guichet(K).Etat  = En_Attente) and
            (Le_Guichet(K).Etape = Ne) then
            return K;
         end if;
      end loop;
      return 0;
   end Un_Numero_De_Guichet_Pret_Pour_L_Etape;

   procedure Traiter_Une_Requete(R: in out Type_De_Requete) is
      Ng : Numero_De_Guichet;
   begin
      -- Est-il possible d'approcher ?
      if (Nombre_De_Guichets_Libres = 0) then
         -- Non, il faut attendre qu'une place soit liberee
         Wait(Un_Guichet_Libre);
      end if;
      -- La requete peut etre prise en compte
      -- 1ere etape : Trouver un guichet libre
      Ng := Un_Numero_De_Guichet_Libre;
      Nombre_De_Guichets_Libres := Nombre_De_Guichets_Libres - 1;
      -- On s'installe
      Le_Guichet(Ng).Requete := R;
      Le_Guichet(Ng).Etat    := En_Attente;
      Le_Guichet(Ng).Etape   := 1;
      -- On reveille un eventuel employe (de l'etape 1) en attente
      Signal(Un_Employe(1));
      -- On attend patiemment la fin du traitement
      Wait(Le_Guichet(Ng).Fin_De_Traitement);
      -- C'est TERMINE pour nous !
      -- Avant de partir, on recupere son bien
      R := Le_Guichet(Ng).Requete;
```

```
      -- On libere le guichet occupe
      Le_Guichet(Ng).Etat := Libre;
      Nombre_De_Guichets_Libres := Nombre_De_Guichets_Libres  + 1;
      -- On reveille un eventuel client en attente d'un guichet libre
      Signal(Un_Guichet_Libre);
   end Traiter_Une_Requete;

   procedure Prendre_Une_Requete(R:  out Type_De_Requete;
                        Ne: in  Numero_D_Etape; Ng: out Numero_De_Guichet) is
   begin
      -- Chercher s'il existe un guichet disposant d'une requete
      --  en attente de cette etape NE
      Ng:= Un_Numero_De_Guichet_Pret_Pour_L_Etape(Ne);
      if Ng = 0 then
         -- Non, personne n'a besoin de nous !
         -- Alors on va attendre !
         Wait(Un_Employe(Ne));
         -- Ca y est !
         -- Un client a besoin de nos services
         -- Rechercher le guichet interesse
         Ng := Un_Numero_De_Guichet_Pret_Pour_L_Etape(Ne);
      end if;
      -- On se sert
      R:= Le_Guichet(Ng).Requete;
      -- et on marque notre passage
      Le_Guichet(Ng).Etat := En_Cours;
   end Prendre_Une_Requete;

   procedure Deposer_Une_Requete(
         R:  in Type_De_Requete;
         Ng: in Numero_De_Guichet) is

   begin
      Le_Guichet(Ng).Requete := R;
      if (Le_Guichet(Ng).Etape = Nombre_D_Etapes)  then
         -- On doit reveiller le client du guichet NG
         Signal(Le_Guichet(Ng).Fin_De_Traitement);
      else
         -- On doit preparer l'etape suivante
         Le_Guichet(Ng).Etape := Le_Guichet(Ng).Etape + 1;
         Le_Guichet(Ng).Etat  := En_Attente;
         -- On doit reveiller un eventuel employe specialiste
         --  de l'etape suivante
         Signal(Un_Employe(Le_Guichet(Ng).Etape));
      end if;
   end Deposer_Une_Requete;

begin
   -- Initialisation du moniteur
   -- Tous les guichets sont libres
   for I in Numero_De_Guichet loop
      Le_Guichet(I).Etat := Libre;
   end loop;
   Nombre_De_Guichets_Libres := Nombre_De_Guichets;
```

```
end Moniteur_De_Gestion_De_Requetes;
```