Le lecteur rédacteur est un problème de synchronisation qui suppose que l'on peut accéder aux données partagées de 2 manières :

— en lecture, c'est a dire sans possibilité de modification ;
— en écriture.

Il est simple de voir que les lectures peuvent être faites simultanément. Par contre, on ne doit pas permettre à deux threads d'être en état de modifier les données, ni permettre la modification en même temps que la lecture.

Le but de ce cours est de construire pas à pas une solution à ce problème.

1 Entête des fonctions

Les primitives de synchronisations sont basées sur 4 fonctions :

— Deux fonctions sont nécessaires pour la lecture :
  — pour signaler/demander le début de lecture ;
  — pour signaler la fin de la lecture.
— Deux fonctions équivalentes sont nécessaires l'écriture
  — pour signaler/demander le début de l'écriture ;
  — pour signaler la fin.

De plus, ces fonctions manipuleront une *variable de synchronisation* commune. Cette variable permet de gérer l'accès aux données manipulées. Cette variable sera un objet complexe contenant les *compteurs*, *mutex* et *variables de condition* nécessaires à son fonctionnement. Le but de cette variable est de pouvoir décider ou non de l'accès au donnée pour le thread qui en fait la demande. Concrètement, c'est en fonction du contenu de cette variable que les fonctions de début de lecture ou de début d'écriture seront blocantes ou pas.

Cela donne donc :

— Une structure :

```
typedef struct LectRed {
  ...// à définir
} LectRed
```

— 2 fonctions d'initialisation et de destruction

```
void LR_init(LectRed *m);
void LR_destroy(LectRed *m);
```

— 4 fonctions pour l'utilisation

```
void LR_demande_lecture(LectRed *m);
void LR_fin_lecture(LectRed *m);
void LR_demande_ecriture(LectRed *m);
void LR_fin_ecriture(LectRed *m);
```

2 Compter les lecteurs et les rédacteurs

Pour faire cela, le premier problème à résoudre est de compter les lecteurs et les rédacteurs. Cela se fait très simplement en incrémentant ou décrémentant

une variable compteur. Mais, il faut faire attention à protéger cette variable par des *muttex*.

La structure LectRed doit donc au moins contenir 2 compteurs (un pour chaque) et 1 muttex (pour protéger les compteurs).

```
typedef struct LectRed {
  int nb_lect;
  int nb_red;
  pthread_mutex_t mut;
} LectRed
```

Les fonctions doivent permettre de compter le nombre de threads en section critique.

```
void LR_demande_lecture(LectRed *m) {
int res;

res = pthread_mutex_lock(&(m->mut));
if (res != 0) {
  perror("LR_demande_lecture : ");
  exit(1);
}
m->nb_lect ++;
res = pthread_mutex_unlock(&(m->mut));
if (res != 0) {
  perror("LR_demande_lecture : ");
  exit(1);
}
}
void LR_fin_lecture(LectRed *m) {
int res;

res = pthread_mutex_lock(&(m->mut));
if (res != 0) {
  perror("LR_fin_lecture : ");
  exit(1);
}
m->nb_lect --;
res = pthread_mutex_unlock(&(m->mut));
if (res != 0) {
  perror("LR_fin_lecture : ");
  exit(1);
}
}

void LR_demande_ecriture(LectRed *m) {
int res;

res = pthread_mutex_lock(&(m->mut));
if (res != 0) {
  perror("LR_demande_ecriture : ");
```

```
    exit(1);
  }

  m->nb_red ++;
  res = pthread_mutex_unlock(&(m->mut));
  if (res != 0) {
    perror("LR_demande_ecriture : ");
    exit(1);
  }
}


void LR_fin_ecriture(LectRed *m) {
  int res;

  res = pthread_mutex_lock(&(m->mut));
  if (res != 0) {
    perror("LR_fin_ecriture : ");
    exit(1);
  }

  m->nb_red --;
  res = pthread_mutex_unlock(&(m->mut));
  if (res != 0) {
    perror("LR_fin_ecriture : ");
    exit(1);
  }

}
```

3 Attente

L'algorithme est alors simple :

- Pour les lecteurs :
  - S'il y a un rédacteur alors attendre
  - Sinon passer
- Pour les rédacteurs
  - S'il y a un lecteur alors attendre.
  - S'il y a un rédacteur alors attendre.
  - Sinon passer.

De même pour sortir de la section critique, le thread doit réveiller ceux qui sont en attente :

- Un lecteur ne doit réveiller les rédacteurs en attente que s'il est le dernier thread à sortir.
- Un rédacteur doit réveiller soit les lecteurs en attente, soit un autre rédacteur.

Pour le cas du rédacteur il faut être capable de savoir s'il y a un lecteur en attente ou alors réveiller systématiquement les deux. Le deuxième cas est plus simple, mais il oblige à re-tester systématiquement la présence de threads en section critique.

```
void LR_demande_lecture(LectRed *m) {
  int res;

  res = pthread_mutex_lock(&(m->mut));
  if (res != 0) {
    perror("LR_demande_lecture : ");
    exit(1);
  }
  while (m->nb_red != 0) {
    // la boucle while sert à  retester l'attente en sortant.
    // Elle est nécessaire car les rédacteurs réveillent systématiquement
    // les lecteurs et les rédacteurs
    pthread_cond_wait(&(m->attente_lect), &(m->mut));
  }
  m->nb_lect ++;

  res = pthread_mutex_unlock(&(m->mut));
  if (res != 0) {
    perror("LR_demande_lecture : ");
    exit(1);
  }
}
void LR_fin_lecture(LectRed *m) {
  int res;

  res = pthread_mutex_lock(&(m->mut));
  if (res != 0) {
    perror("LR_fin_lecture : ");
    exit(1);
  }
  m->nb_lect --;
  if (m->nb_lect == 0) {
    // C'est le dernier thread en lecture, on réveille les rédacteurs en
    // attente.
    pthread_cond_signal(&(m->attente_red));
  }
  res = pthread_mutex_unlock(&(m->mut));
  if (res != 0) {
    perror("LR_fin_lecture : ");
    exit(1);
  }
}

void LR_demande_ecriture(LectRed *m) {
  int res;
```

```
  res = pthread_mutex_lock(&(m->mut));
  if (res != 0) {
    perror("LR_demande_ecriture : ");
    exit(1);
  }


  while (m->nb_red != 0 || m->nb_lect != 0) {
     pthread_cond_wait(&(m->attente_red), &(m->mut));
  }

  m->nb_red ++;

  res = pthread_mutex_unlock(&(m->mut));
  if (res != 0) {
    perror("LR_demande_ecriture : ");
    exit(1);
  }
}


void LR_fin_ecriture(LectRed *m) {
  int res;

  res = pthread_mutex_lock(&(m->mut));
  if (res != 0) {
    perror("LR_fin_ecriture : ");
    exit(1);
  }

  m->nb_red --;
  pthread_cond_broadcast(&(m->attente_lect));
  pthread_cond_signal(&(m->attente_red));

  res = pthread_mutex_unlock(&(m->mut));
  if (res != 0) {
    perror("LR_fin_ecriture : ");
    exit(1);
  }

}
```

4 Famine

Que se passe-t'il s'il y a des lecteurs qui arrivent régulièrement ?

Dans ce cas, les rédacteurs ne peuvent jamais obtenir le passage. En effet, si un rédacteur est en attente, les lecteurs qui arrivent lui passent devant et

l'empêchent d'accéder aux données. Pour résoudre le problème, il faut empêcher l'arrivée de nouveaux lecteurs dès qu'un rédacteur est en attente. Un bon moyen pour cela est d'utiliser un *mutex* partagé par les lecteurs et les rédacteurs pendant leur fonction de demande d'accès. Ainsi, si un thread est bloqué dans cette fonction, plus aucun autre thread ne peut y accéder. Ce *mutex* permet alors de conserver l'ordre d'arrivée dans la file d'attente.

Cela donne ce fichier de code :

```
// compiler avec
// gcc -g -Wall  -D_REENTRANT -D_GNU_SOURCE lecteur_redacteur.c
//                                  -lm -lpthread -o lectrect
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>

#include <pthread.h>
#include <errno.h>
#include <string.h>

/*************************************/
/*                                   */
/*             entete                */
/*                                   */
/*************************************/
typedef struct {
  int num;
  int seed;
  int nbtours;
} param_t;

typedef struct {
  int nb_lect;
  int nb_red;

  pthread_mutex_t mut;
  pthread_mutex_t mut_ordre;

  pthread_cond_t attente_lect;
  pthread_cond_t attente_red;
} lect_red_t;

lect_red_t* lect_red_init();
void lect_red_destroy(lect_red_t *m);
void lect_red_demande_lecture(lect_red_t *m);
void lect_red_fin_lecture(lect_red_t *m);
void lect_red_demande_ecriture(lect_red_t *m);
void lect_red_fin_ecriture(lect_red_t *m);

/* #include "lecteur_redacteur.h" */
```

```
/*************************************/
/*                                   */
/*               code                */
/*                                   */
/*************************************/
lect_red_t *m;

lect_red_t* lect_red_init() {
  int res;
  lect_red_t *m = malloc(sizeof(lect_red_t));

  // Initialisation des compteurs à 0
  m->nb_lect = 0;
  m->nb_red = 0;

  // Initialisation des mutex sans attributs
  res = pthread_mutex_init(&(m->mut), NULL);
  if (res != 0) {
    fprintf(stderr, "lect_red_init : %s", strerror(res));
    exit(1);
  }
  res = pthread_mutex_init(&(m->mut_ordre), NULL);
  if (res != 0) {
    fprintf(stderr, "lect_red_init : %s", strerror(res));
    exit(1);
  }


  // Initialisation des variables conditionnelles
  res = pthread_cond_init(&(m->attente_lect), NULL);
  if (res != 0) {
    fprintf(stderr, "lect_red_init : %s", strerror(res));
    exit(1);
  }
  res = pthread_cond_init(&(m->attente_red), NULL);
  if (res != 0) {
    fprintf(stderr, "lect_red_init : %s", strerror(res));
    exit(1);
  }

  return m;
}

void lect_red_destroy(lect_red_t *m) {
  int res;

  // Récupération des ressources des mutex
  res = pthread_mutex_destroy(&(m->mut));
  if (res != 0) {
```

```
    fprintf(stderr, "lect_red_destroy : %s", strerror(res));
    exit(1);
  }
  res = pthread_mutex_destroy(&(m->mut_ordre));
  if (res != 0) {
    fprintf(stderr, "lect_red_destroy : %s", strerror(res));
    exit(1);
  }

  res = pthread_cond_destroy(&(m->attente_lect));
  if (res != 0) {
    fprintf(stderr, "lect_red_destroy : %s", strerror(res));
    exit(1);
  }

  res = pthread_cond_destroy(&(m->attente_red));
  if (res != 0) {
    fprintf(stderr, "lect_red_destroy : %s", strerror(res));
    exit(1);
  }

  free(m);
}

void lect_red_demande_lecture(lect_red_t *m) {
  int res;


  res = pthread_mutex_lock(&(m->mut));
  if (res != 0) {
    fprintf(stderr, "lect_red_demande_lecture : %s", strerror(res));
    exit(1);
  }
  res = pthread_mutex_lock(&(m->mut_ordre));
  if (res != 0) {
    fprintf(stderr, "lect_red_demande_lecture : %s", strerror(res));
    exit(1);
  }

  while (m->nb_red != 0) {
    // la boucle while sert à retester l'attente en sortant
    // Elle est nécessaire car les redacteur réveillent systématiquement
    // les lecteur et les rédacteurs
    pthread_cond_wait(&(m->attente_lect), &(m->mut));
  }
  m->nb_lect ++;

  res = pthread_mutex_unlock(&(m->mut));
  if (res != 0) {
    fprintf(stderr, "lect_red_demande_lecture : %s", strerror(res));
```

```
    exit(1);
  }
  res = pthread_mutex_unlock(&(m->mut_ordre));
  if (res != 0) {
    fprintf(stderr, "lect_red_demande_lecture : %s", strerror(res));
    exit(1);
  }
}

void lect_red_fin_lecture(lect_red_t *m) {
  int res;

  res = pthread_mutex_lock(&(m->mut));
  if (res != 0) {
    fprintf(stderr, "lect_red_fin_lecture : %s", strerror(res));
    exit(1);
  }
  m->nb_lect --;
  if (m->nb_lect == 0) {
    // C'est le dernier thread en lecture, on réveille les rédacteurs en
    // attente.
    pthread_cond_signal(&(m->attente_red));
  }
  res = pthread_mutex_unlock(&(m->mut));
  if (res != 0) {
    fprintf(stderr, "lect_red_fin_lecture : %s", strerror(res));
    exit(1);
  }
}

void lect_red_demande_ecriture(lect_red_t *m) {
  int res;


  res = pthread_mutex_lock(&(m->mut));
  if (res != 0) {
    fprintf(stderr, "lect_red_demande_ecriture : %s", strerror(res));
    exit(1);
  }
  res = pthread_mutex_lock(&(m->mut_ordre));
  if (res != 0) {
    fprintf(stderr, "lect_red_demande_ecriture : %s", strerror(res));
    exit(1);
  }

  while (m->nb_red != 0 || m->nb_lect != 0) {
     pthread_cond_wait(&(m->attente_red), &(m->mut));
  }

  m->nb_red ++;
```

```
  res = pthread_mutex_unlock(&(m->mut));
  if (res != 0) {
    fprintf(stderr, "lect_red_demande_ecriture :  %s", strerror(res));
    exit(1);
  }
  res = pthread_mutex_unlock(&(m->mut_ordre));
  if (res != 0) {
    fprintf(stderr, "lect_red_demande_ecriture :  %s", strerror(res));
    exit(1);
  }

}

void lect_red_fin_ecriture(lect_red_t *m) {
  int res;

  res = pthread_mutex_lock(&(m->mut));
  if (res != 0) {
    fprintf(stderr, "lect_red_fin_ecriture :  %s", strerror(res));
    exit(1);
  }

  m->nb_red --;
  pthread_cond_broadcast(&(m->attente_lect));
  pthread_cond_signal(&(m->attente_red));

  res = pthread_mutex_unlock(&(m->mut));
  if (res != 0) {
    fprintf(stderr, "lect_red_fin_ecriture :  %s", strerror(res));
    exit(1);
  }

}

void *lecture(param_t *a) {
  int i;
  srand(a->seed);

  for (i=0;i<a->nbtours; i++) {
    lect_red_demande_lecture(m);
    printf("rlecteur \tnum %d\ttour %d : en section critique %d\tlect %d\tred\n", a->nu
    usleep((int) ceil(rand()*100.0/RAND_MAX*1.0));
    lect_red_fin_lecture(m);
    usleep((int) ceil(rand()*100.0/RAND_MAX*1.0));
  }

  return NULL;
}
```

```
void *ecriture(param_t *a) {
  int i;
  srand(a->seed);

  for (i=0;i<a->nbtours; i++) {
    lect_red_demande_ecriture(m);
    printf("redacteur \tnum %d\ttour %d : en section critique %d\tlect %d\tred\n", a->r
    usleep((int) ceil(rand()*100.0/RAND_MAX*1.0));
    lect_red_fin_ecriture(m);
    usleep((int) ceil(rand()*100.0/RAND_MAX*1.0));
  }

  return NULL;
}

int main(int argc, char *argv[]) {
  int nbl, nbr, nbt, i;
  int res;

  if (
      (argc != 4)
      || ((nbl=atoi(argv[1]))==0)
      || ((nbr=atoi(argv[2]))==0)
      || ((nbt=atoi(argv[3]))==0)
      )
    {
      fprintf(stderr, "usage %s <nblect> <nbred> <nbtours>\n", argv[0]);
      exit(1);
    }

  // tableau qui contiendra les arguement au threads
  param_t args[nbl+nbr];
  // tableua qui contiendra les numéro de threads (pour fair un join)
  pthread_t tab[nbl+nbr];

  m = lect_red_init();

  for (i=0; i<nbl; i++) {
    args[i].num=i;
    args[i].seed = rand();
    args[i].nbtours = nbt;
    res  = pthread_create(&tab[i], NULL,(void * (*)(void *)) lecture, &args[i]);
    if (res) {
      fprintf(stderr, "create !! %s\n", strerror(res));
      exit(1);
    }
  }
  for (i=0; i<nbr; i++) {
    args[i+nbl].num=i;
    args[i+nbl].seed = rand();
```

```
    args[i].nbtours = nbt;
    res  = pthread_create(&tab[i+nbl], NULL,(void * (*)(void *)) ecriture, &args[i]);
    if (res) {
      fprintf(stderr, "create !! %s\n", strerror(res));
      exit(1);
    }
  }


  for (i=0; i<nbl+nbr; i++) {
    res = pthread_join(tab[i], NULL);
    if (res) {
      fprintf(stderr, "join !! %s\n", strerror(res));
      exit(1);
    }
  }

  return 0;
}
```