Compte-rendu TP "Réalisation d'un mini noyau temps réel - Partie 3 et 4"

Mewen Michel et Sander Ricou - MI11 UTC

Partie 3 : exclusion mutuelle

Questions:
*On nous demande d'ajouter au mini-noyau temps réel la gestion de la suspension et du réveil d'une tâche.
On a donc réalisé les deux primitives suivantes :

- `void dort(void)` : endort (suspend) la tâche courante
- `void reveille(register ushort tache)` : réveille la tâche tache. Le signal de réveil n'est pas mémorisé si la tâche n'est pas suspendue.
  Vous modifierez les fichiers NOYAU.C et NOYAU.H afin d'implémenter ces deux primitives.*

Voici le code de ces fonctions :

```
void reveille(uint16_t t)
{
    CONTEXTE *p;
    //récuparation du contexte général
    p = &_contexte[t];

    // Erreur si tâche non créée
    if (p->status == NCREE)
    noyau_exit();

    // Protection des accès sur la file
    _lock_();
    if (p->status == SUSP) {
        p->status = EXEC;

        // Ajout de la tâche précédement endormie à la file des tâches à
exécuter
        ajoute(t);

        // Appel de la tâche suivante par le scheduler
        schedule();
    }
    _unlock_();
}
```

```
void dort(void)
{
    CONTEXTE *p;
    p = &_contexte[_tache_c];

    if (p->status == NCREE)
        noyau_exit();

    _lock_();
    if (p->status == EXEC) {
        p->status = SUSP;
        retire(_tache_c);
        // Appel de la tâche suivante par le scheduler
        schedule();
    }
    _unlock_();

}
```

Quand on suspend une tâche, on la retire donc de la file des tâches à exécuter et on la rajoute au réveil. Il ne faut pas oublier les appels à la fonction `schedule()` permettant de lancer la tâche suivante du scheduler. De plus nous vérifions bien que la tâche est en execution pour l'endormir et inversement suspendu pour la réveiller. Si la tâche n'est pas créée alors on interromp le noyau. Notons également l'utilisation des primitives `_lock_()` et `_unlock_()` qui évitent les péemptions pendant l'exécution du code entre ces deux fonctions, aucune interruption n'est acceptée. Ceci nous assure la protection de la section critique.

On réalise maintenant un producteur et un consommateur selon les besoins définis dans le sujet :

Le producteur va s'endormir si le nombre de places vides est nul et il va réveiller le consommateur s'il y a un seul élément dans la file. Le consommateur va s'endormir si le nombre places vides est égal à la taille de la file et va réveiller le producteur dès qu'il y a une case vide. Il peut donc y avoir les 2 tâches lancées en même temps mais elles ne peuvent pas être éteintes en même temps !

On va donc s'inspirer de notre file du TP précédent pour que les deux tâches se transmettent des entiers. En dehors de cette communication, on va les occuper pour simuler la préparation et le traitement des données.

Nos variables globales:

```
#define NB_LOOPS    20
#define PROD_wait_OLD   100000
#define CONS_wait_OLD   100000
```

1ère version:

Le producteur

Pendant `NB_LOOPS` itérations le producteur va produire un entier n dans la file circulaire (`file_circ_push(n)`), il affiche ensuite la file. Puis si celle-ci est pleine, il s'endort et si il remarque que la file à au moins un élément alors il réveille le consommateur avec `reveille(2)` (tâche identifiée par 2, car créée en second). A la fin des itérations le producteur appelle `fin_tache()`, la tâche est alors interrompue.

Voici notre tâche producteur :

```
TACHE   producteur(void)
{
  puts("------> DEBUT producteur");

  for (int n = 0; n < NB_LOOPS; n++) {
      wait(PROD_wait);
      int res = file_circ_push ( n );
      if (res == -1) {
          puts("Erreur : la file est pleine !");
      } else {
          printf("Producteur a ecrit %d\n", res);
      }
      affic_file_circ();

      if (file_full()) {
          puts("Prod s'endort");
          dort();
      }

      if (file_nearly_empty()) {
          puts("Prod reveille cons");
          reveille(2);
      }
  }
  puts("------> FIN producteur");
  fin_tache();
}
```

Le consommateur

Le consommateur est endormi des sa création, il sera réveiller par le producteur quand celui-ci aura produit un premier élément dans la file circulaire. Des qu'il est réveillé il se met à consommer les entiers de la file.
La file est à nouveau afficher et le consommateur teste si la file est vide, si c'est la cas il s'endort. Des qu'il s'aperçoit que la file à un premier élément vide il réveille alors le producteur (première tâche créée d'où le numéro 1), `reveille(1)`.

Voici notre tâche consommateur:

```
TACHE   consommateur(void)
{
  puts("------> DEBUT consommateur");
  puts("Cons s'endort");
  dort();
  while (1) {
      wait(CONS_wait);
      int res = file_circ_pop ();

      if (res == -1) {
          puts("Erreur : la file est vide");
      } else {
          printf("Consommateur a lu %d\n", res);
      }
      affic_file_circ();

      if (file_empty()) {
          puts("Cons s'endort");
          dort();
      }
      if (file_nearly_full()) {
          puts("Cons reveille prod");
          reveille(1);
      }
  }
}
```

2ème version:

Le producteur

Pendant NB_LOOPS itérations le producteur va produire un entier n dans la file circulaire (`file_circ_push(n)`), il affiche ensuite la file. Puis si celle-ci est pleine, il s'endort et si il remarque que la file à au moins un élément alors il réveille le consommateur avec `reveille(2)` (tâche identifiée par 2, car créée en second). A la fin des itérations le producteur appelle `fin_tache()`, la tâche est alors interrompue.

Voici notre tâche producteur :

```
TACHE   producteur(void)
{
  puts("------> DEBUT producteur");

  for (int n = 0; n < NB_LOOPS; n++) {
      wait(PROD_wait);
      int res = file_circ_push ( n );
      if (res == -1) {
          puts("Erreur : la file est pleine !");
      } else {
          printf("Producteur a ecrit %d\n", res);
      }
      affic_file_circ();

      if (file_full()) {
          puts("Prod reveille cons");
          reveille(2);

          puts("Prod s'endort");
          dort();
      }
  }
  puts("------> FIN producteur");
  fin_tache();
}
```

Le consommateur

Cette fois-ci quand le consommateur s'aperçoit que la file est vide alors il réveille le producteur et s'endort.

Voici la deuxième version du consomateur:

```
TACHE   consommateur(void)
{
  puts("------> DEBUT consommateur");
  puts("Cons s'endort");
  dort();
  while (1) {
      wait(CONS_wait);
      int res = file_circ_pop ();

      if (res == -1) {
          puts("Erreur : la file est vide");
      } else {
          printf("Consommateur a lu %d\n", res);
      }
      affic_file_circ();

      if (file_empty()) {
          puts("Cons reveille prod");
          reveille(1);

          puts("Cons s'endort");
          dort();
      }
  }
}
```

Note sur FIFO circulaire implémentée

La file ciculaire utilisée dans le producteur et le consommateur est initialisée dans notre main avec la fonction `fifo_circ_init()`.
Nous allons ci-dessous, vous détaillez les différentes fonctions nous permettant de gérer notre file circulaire.

La fonction suivante nous permet d'initialiser la file avec que des éléments vides:

```
void fifo_circ_init(FIFO* f)
{
    f->tete = 0;
    f->queue = 0;
    for(int i = 0; i < NB_MAX; i++) {
        f->fifo[i] = FILE_VIDE;
    }
}
```

Pour savoir si la file est vide, la fonction suivante teste si la queue pointe sur un élément vide, si c'est le cas alors la file est vide.
En effet ceci correspond au premier élément de notre file.

```
char fifo_empty(FIFO* f) {
    if (f->fifo[f->queue] == FILE_VIDE) {
        //printf("File vide, f->queue = %d, f[q] = %d, fifo_VIDE = %d.", f-
>queue, _fifo_mutex[f->queue], fifo_VIDE);
        return 1;
    } else {
        return 0;
    }
}
```

Sur le même principe que précédemment, nous allons tester si la tête n'est pas vide, le cas échéant la file est pleine:

```
char fifo_full(FIFO* f) {
    if (f->fifo[f->tete] != FILE_VIDE) {
        return 1;
    } else {
        return 0;
    }
}
```

Les deux fonction suivantes nous permettent d'endormir et réveiller les tâches au moment oportun afin de garantir l'exclusion mutuelle.

Notons que la queue pointe sur le prochain élément à traiter et que la tête pointe sur l'emplacement où le prochain élement entrant sera stocké.

Cette fonction teste si la file est presque pleine. Pour être dans cette situation soit la tête est juste une case derrière la queue, soit en particulier quand la queue vaut 0 (pointe sur la première case du tableau, le -1 n'a plus de sens) alors tête a pour valeur le nombre d'élement au maximum de notre file - 1.

```
char    fifo_nearly_full(FIFO* f)
{
    return (f->tete == (f->queue - 1))
            || ((f->tete == (NB_MAX - 1))
                    && (f->queue == 0));
}
```

On retrouve la réciproque pour savoir si le tableau est presque vide, soit la tête est positionner une case devant la queue, soit dans le cas particulier où la tête est égale 0 alors la tête (ne pouvant pas valoir -1) aura en valeur le nombre d'élement au maximum de notre file - 1 (dernière emplacement de notre file).

```
char    fifo_nearly_empty(FIFO* f)
{
    return (f->tete == (f->queue + 1))
            || ((f->queue == (NB_MAX - 1))
                    && (f->tete == 0));
}
```

Cette fonction retourne l'id suivante de l'id passé en argument, si nous sommes en bout de file alors elle retourne zéro.

```
uint16_t fifo_next_idx(FIFO* f, uint16_t idx) {
    if(idx+1 < NB_MAX) {
        return idx+1;
    } else {
        return 0;
    }
}
```

La fonction push, insére l'élément au niveau du pointeur "tête" de notre file.

```
int16_t fifo_circ_push (FIFO* f, uint16_t n )
{
    if(!fifo_full(f)) {
        f->fifo[f->tete] = n;
        f->tete = fifo_next_idx(f, f->tete);
        return n;
    } else {
        return -1;
    }
}
```

La fonction pop quant à elle permet de retire l'élément le plus ancien de la file, l'élément pointé par "queue".

```
int16_t fifo_circ_pop(FIFO* f)
{
    int16_t res = -1;
    if(!fifo_empty(f)) {
        res = f->fifo[f->queue];

        f->fifo[f->queue] = FILE_VIDE;

        f->queue = fifo_next_idx(f, f->queue);
    }

    return res;
}
```

Et enfin c'est fonction affiche simplement le contenu de notre file.

```
void affic_fifo_circ(FIFO* f)
{
    if(fifo_empty(f)){
        puts("Fifo vide");
        return;
    }

    uint16_t  idx = f->queue;
    while (idx != f->tete) {
        printf("[%d] -> %d\n", idx, f->fifo[idx]);
        idx = fifo_next_idx(f, idx);
    }
}
```

TEST du MUTEX:

Voici les résultats obtenus avec nos deux versions.

1ère version:

Nous avons effectué trois tests différents en faisant varier les temps de traitement du producteur et du consommateur.

Avec les paramètres suivants:
Le producteur est plus rapide que le consommateur.

- 'PROD_WAIT 100000'
- 'CONS_WAIT 1000000'

```
Test noyau MUTEX
------> EXEC tache A
------> DEBUT producteur
------> DEBUT consommateur
Cons s'endort
Producteur a ecrit 0
[0] -> 0
Prod reveille cons
Producteur a ecrit 1
[0] -> 0
[1] -> 1
Producteur a ecrit 2
[0] -> 0
[1] -> 1
[2] -> 2

...

[0] -> 0
[1] -> 1
[2] -> 2
[3] -> 3
[4] -> 4
[5] -> 5
Producteur a ecrit 6
[0] -> 0
[1] -> 1
[2] -> 2
[3] -> 3
[4] -> 4
[5] -> 5
[6] -> 6
Producteur a ecrit 7
Prod s'endort
Consommateur a lu 0
[1] -> 1
[2] -> 2
```

```
[3] -> 3
[4] -> 4
[5] -> 5
[6] -> 6
[7] -> 7
Cons reveille prod
Producteur a ecrit 8
Prod s'endort
Consommateur a lu 1
[2] -> 2
[3] -> 3
[4] -> 4
[5] -> 5
[6] -> 6
[7] -> 7
[0] -> 8
Cons reveille prod
Producteur a ecrit 9
Prod s'endort
Consommateur a lu 2
[3] -> 3
[4] -> 4
[5] -> 5
[6] -> 6
[7] -> 7
[0] -> 8
[1] -> 9

...

Cons reveille prod
Producteur a ecrit 19
Prod s'endort
Consommateur a lu 12
[5] -> 13
[6] -> 14
[7] -> 15
[0] -> 16
[1] -> 17
[2] -> 18
[3] -> 19
Cons reveille prod
------> FIN producteur
Consommateur a lu 13
[6] -> 14
[7] -> 15
[0] -> 16
[1] -> 17
[2] -> 18
[3] -> 19
Consommateur a lu 14
[7] -> 15
```

```
[0] -> 16
[1] -> 17
[2] -> 18
[3] -> 19

...

Consommateur a lu 17
[2] -> 18
[3] -> 19
Consommateur a lu 18
[3] -> 19
Consommateur a lu 19
Plus rien \0xef\0xbf\0xbd ordonnancer.
Sortie du noyau

Activations tache 0 : 3
Activations tache 1 : 99
Activations tache 2 : 896
Activations tache 3 : 0
Activations tache 4 : 0
Activations tache 5 : 0
Activations tache 6 : 0
Activations tache 7 : 0
```

Avec les paramètres suivants:
Le producteur est plus lent que le consommateur.

- 'PROD_WAIT 1000000'
- 'CONS_WAIT 100000'

```
Test noyau MUTEX
------> EXEC tache A
------> DEBUT producteur
------> DEBUT consommateur
Cons s'endort
Producteur a ecrit 0
[0] -> 0
Prod reveille cons
Consommateur a lu 0
Fifo vide
Cons s'endort
Producteur a ecrit 1
[1] -> 1
Prod reveille cons
Consommateur a lu 1
Fifo vide
Cons s'endort

...

Producteur a ecrit 18
[2] -> 18
Prod reveille cons
Consommateur a lu 18
Fifo vide
Cons s'endort
Producteur a ecrit 19
[3] -> 19
Prod reveille cons
------> FIN producteur
Consommateur a lu 19
Fifo vide
Cons s'endort
Plus rien \0xef\0xbf\0xbd ordonnancer.
Sortie du noyau

Activations tache 0 : 3
Activations tache 1 : 897
Activations tache 2 : 102
Activations tache 3 : 0
Activations tache 4 : 0
Activations tache 5 : 0
Activations tache 6 : 0
Activations tache 7 : 0
```

Avec les paramètres suivants:
Le producteur et le consommateur vont à la même vitesse.

- 'PROD_WAIT 100000'
- 'CONS_WAIT 100000'

```
Test noyau MUTEX
------> EXEC tache A
------> DEBUT producteur
------> DEBUT consommateur
Cons s'endort
Producteur a ecrit 0
[0] -> 0
Prod reveille cons
Consommateur a lu 0
Fifo vide
Cons s'endort
Producteur a ecrit 1
[1] -> 1
Prod reveille cons
Consommateur a lu 1
Fifo vide
Cons s'endort

...

Producteur a ecrit 18
[2] -> 18
Prod reveille cons
Consommateur a lu 18
Fifo vide
Cons s'endort
Producteur a ecrit 19
[3] -> 19
Prod reveille cons
------> FIN producteur
Consommateur a lu 19
Fifo vide
Cons s'endort
Plus rien \0xef\0xbf\0xbd ordonnancer.
Sortie du noyau

Activations tache 0 : 3
Activations tache 1 : 121
Activations tache 2 : 101
Activations tache 3 : 0
Activations tache 4 : 0
Activations tache 5 : 0
Activations tache 6 : 0
Activations tache 7 : 0
```

En conclusion, au travers de ces différents tests nous observons bien que la synchronisation des tâches est assurée par notre implémentation, cependant sur cette version nous pouvons observer des accès concurrent à la section critique (notre file circulaire).
La seconde version quant à elle assure que seulement une tâche à la fois accède à la file circulaire et répond ainsi entièrement au sujet du TP.

2ème version

Nous avons effectué trois tests différents en faisant varier les temps de traitement du producteur et du consommateur.
Cependant ceci n'est pas visible dans les logs (il aurait fallu analyser les temps de traitement), les différences se verront sur les durées des phases de production et de consommation.

```
Test noyau MUTEX
------> EXEC tache A
------> DEBUT producteur
------> DEBUT consommateur
Cons s'endort
Producteur a ecrit 0
[0] -> 0
Producteur a ecrit 1
[0] -> 0
[1] -> 1
Producteur a ecrit 2
[0] -> 0
[1] -> 1
[2] -> 2

...


Producteur a ecrit 6
[0] -> 0
[1] -> 1
[2] -> 2
[3] -> 3
[4] -> 4
[5] -> 5
[6] -> 6
Producteur a ecrit 7
[0] -> 0
[1] -> 1
[2] -> 2
[3] -> 3
[4] -> 4
[5] -> 5
[6] -> 6
[7] -> 7
Prod reveille cons
Prod s'endort
Consommateur a lu 0
[1] -> 1
[2] -> 2
[3] -> 3
[4] -> 4
[5] -> 5
[6] -> 6
[7] -> 7
Consommateur a lu 1
[2] -> 2
```

```
[3] -> 3
[4] -> 4
[5] -> 5
[6] -> 6
[7] -> 7

...

Consommateur a lu 7
Fifo vide
Cons reveille prod
Cons s'endort
Producteur a ecrit 8
[0] -> 8
Producteur a ecrit 9
[0] -> 8
[1] -> 9
...

Producteur a ecrit 15
[0] -> 8
[1] -> 9
[2] -> 10
[3] -> 11
[4] -> 12
[5] -> 13
[6] -> 14
[7] -> 15
Prod reveille cons
Prod s'endort
Consommateur a lu 8
[1] -> 9
[2] -> 10
[3] -> 11
[4] -> 12
[5] -> 13
[6] -> 14
[7] -> 15
Consommateur a lu 9
[2] -> 10
[3] -> 11
[4] -> 12
[5] -> 13
[6] -> 14
[7] -> 15

...

Consommateur a lu 15
Fifo vide
Cons reveille prod
Cons s'endort
```

```
Producteur a ecrit 16
[0] -> 16
Producteur a ecrit 17
[0] -> 16
[1] -> 17
...

Producteur a ecrit 19
[0] -> 16
[1] -> 17
[2] -> 18
[3] -> 19
Prod reveille cons
Prod s'endort
Consommateur a lu 16
[1] -> 17
[2] -> 18
[3] -> 19

...

Consommateur a lu 19
Fifo vide
Cons reveille prod
------> FIN producteur
Cons s'endort
Plus rien \0xef\0xbf\0xbd ordonnancer.
Sortie du noyau
```

Ces différents résultats nous montre bien qu'aucun accès concurrent n'est permis par notre code, l'accès à la section critique est correctement assurer pas notre exclusion mutuelle.

Partie 4 :sémaphores

Dans la partie 3 du TP, on a vu que la résolution des conflits d'accès à des ressources partagées était un problème complexe et difficile à régler sans plus d'outils. Les sémaphores offrent une solution générale à ces problèmes.

Questions
1 - On vous demande d'ajouter au mini-noyau temps réel la gestion de sémaphores à compte. Vous
placerez dans un fichier SEM.C les structures de données permettant la gestion des sémaphores ainsi que
les diverses fonctions d'accès (primitives), et dans SEM.H les déclarations permettant d'en faire usage
dans un programme utilisateur.

Pour la création de notre sémaphore, nous réutilisons la file circulaire crée et décrite dans la partie précédente. Grâce à cette file nous allons pouvoir gérer le nombre d'accès au sémaphore ainsi que mettre en attente d'éventuelles tâches.

Ci-dessous la fonction initialise le tableau des sémaphores _sem.

```
void s_init( void ) {
    for(short i = 0; i < MAX_SEM; i++) {
        FIFO new_fifo;
        _sem[i].valeur = INIT_SEM;
        _sem[i].file = &new_fifo;
    }
}
```

La fonction suivante crée un sémaphore dans un emplacement libre, retourne le numéro d'index dans _sem du sémaphore créé.
La valeur initiale du compteur est v.

```
ushort s_cree( short v ) {
    short i = 0;
    while (i <= MAX_SEM
            && _sem[i].valeur != INIT_SEM) {
        i++;
    }

    if(i<MAX_SEM) {
        _sem[i].valeur = v;
        fifo_circ_init(_sem[i].file);
        return i;
    } else {
        return MAX_SEM;
    }
}
```

La fonction suivante supprime le sémaphore n.

```
void s_close( ushort n ) {
    if (_sem[n].valeur < 0) {
        _sem[n].valeur = INIT_SEM;
    }
}
```

Implémente P(s). Tente de prendre le sémaphore n.

```
void s_wait( ushort n ){
    //printf("Wait : Le semaphore a pour valeur %d -> --\n", _sem[n].valeur);
    _sem[n].valeur--;

    if (_sem[n].valeur < 0) {
        // Bloquer la tâche et la mettre dans la file
        fifo_circ_push(_sem[n].file, _tache_c);
        //puts("S'endort en attendant le semaphore");
        dort();
    } else {
        //puts("J'ai le semaphore !!! :D");
    }
}
```

Implémente V(s). Libère le semaphore n.

```
void s_signal ( ushort n ) {
    //printf("Signal : Le semaphore a pour valeur %d -> ++\n", _sem[n].valeur);
    _sem[n].valeur++;

    if (_sem[n].valeur <= 0) {
        // Libérer (délivrer) la prochaine tâche de la fifo
        int16_t tacheId = fifo_circ_pop(_sem[n].file);
        //printf("Donne le semaphore au suivant : %d\n", tacheId);
        reveille(tacheId);
    }
}
```

2 – En utilisant les primitives que vous venez de définir, reprendre le modèle producteur/consommateur et résoudre les problèmes d'exclusion mutuelle et de synchronisation entre le producteur et le consommateur en utilisant exclusivement des sémaphores. Vérifiez le fonctionnement avec plusieurs producteurs et consommateurs.

La tâche tache0 est démarrée dans le main et permet d'initialiser NB_PROD producteur et NB_CONS consomamteur.
De plus on crée un sémaphore qui permet de gérer l'accès des consommateurs (prod_to_cons) et un autre pour l'accès des producteurs à la ressource (cons_to_prod). Le nombre de jeton sur les sémaphores est 0, en effet on veut mettre toute tâche voulant accèder à la variable en attente et les libérer ensuite au moment voulu.

```
TACHE   tache0(void)
{
  puts("------> EXEC tache A");
  file_circ_init();
  sem_id_prod_to_cons = s_cree( 0 );
  sem_id_cons_to_prod = s_cree( 0 );
  int i;
  for(i=0; i<NB_PROD; i++){
      active(cree(prod));
  }
  for(i=0; i<NB_CONS; i++){
      active(cree(consomm));
  }
  fin_tache();
}
```

Les producteurs et consomamteurs se comportent de la même manière que dans la partie 3.

Ci-dessous vous trouverez la procédure pour la création d'une tâche producteur. Sur le même principe que précédemment on "endort" le producteur quand la file partagée est pleine (s_wait(sem_id_cons_to_prod)), plus précisement il est mis en attente sur le sémaphore (jeton = 0).
Lorsque que la file est presque vide, c'est le moment de "réveiller" un consommateur (s_signal(sem_id_prod_to_cons)) en lui donnant l'accès au sémaphore.

```
TACHE   prod()
{
    printf("------> DEBUT Prod%d\n", _tache_c);

  for (int n = _tache_c; n < NB_LOOPS; n += NB_PROD) {
      wait(PROD_WAIT);
      int res = file_circ_push ( n );
      if (res == -1) {
          printf("Erreur Prod%d : la file est pleine !\n", _tache_c);
          n -= NB_PROD;
      } else {
          printf("[Prod%d > File] %d\n", _tache_c, res);
      }
      affic_file_circ();

      if (file_full()) {
          printf("Prod%d s'endort\n", _tache_c);
          s_wait(sem_id_cons_to_prod);
          printf("Prod%d se reveille\n", _tache_c);
      }

      if (file_nearly_empty()) {
          printf("Prod%d reveille cons\n", _tache_c);
          s_signal(sem_id_prod_to_cons);
      }
  }
  printf("------> FIN Prod%d\n", _tache_c);
  fin_tache();
}
```

Sous le même principe que la tâche prod(), cosomm() va à l'inverse, quand la file est vide, "endormir" le consommateur (`s_wait(sem_id_prod_to_cons)`) en le mettant en attente au niveau du sémaphore et quand la file est presque pleine permettre à un producteur d'accèder à la variable partagée.

```
TACHE   consomm(void)
{
  printf("------> DEBUT Cons%d\n", _tache_c);
  while (1) {
      wait(CONS_WAIT);
      int res = file_circ_pop ();

      if (res == -1) {
          printf("Erreur Cons%d : la file est vide\n", _tache_c);
      } else {
          printf("[File > Cons%d] %d\n", _tache_c, res);
      }
      affic_file_circ();

      if (file_empty()) {
          printf("Cons%d s'endort\n", _tache_c);
          s_wait(sem_id_prod_to_cons);
      }
      if (file_nearly_full()) {
          printf("Cons%d reveille prod\n", _tache_c);
          s_signal(sem_id_cons_to_prod);
      }
  }
  printf("------> FIN Cons%d\n", _tache_c);
  fin_tache();
}
```

On peut donc grâce aux sémaphores gérer l'accès à la file partagée avec plusieurs consommateurs et plusieurs producteurs. Si plusieurs tâches veulent accèder à la section critique, on laisse ici seulement une tâche faire ses modifications et les autres sont mises en attente.
Les logs ci-dessous mettent en évidence ceci:

Avec 1 producteur et un consommateur :

```
Test noyau Semaphore
------> EXEC tache A
------> DEBUT Prod1
------> DEBUT Cons2
[Prod1 > File] 1
[0] -> 1
Prod1 reveille cons
[Prod1 > File] 2
[0] -> 1
[1] -> 2
[File > Cons2] 1
[1] -> 2
[Prod1 > File] 3
[1] -> 2
[2] -> 3
[File > Cons2] 2
```

```
[2] -> 3
[Prod1 > File] 4
[2] -> 3
[3] -> 4
[Prod1 > File] 5
[2] -> 3
[3] -> 4
[4] -> 5
[Prod1 > File] 6
[2] -> 3
[3] -> 4
[4] -> 5
[5] -> 6
[File > Cons2] 3
[3] -> 4
[4] -> 5
[5] -> 6
[Prod1 > File] 7
[3] -> 4
[4] -> 5
[5] -> 6
[6] -> 7
[File > Cons2] 4
[4] -> 5
[5] -> 6
[6] -> 7
[Prod1 > File] 8
[4] -> 5
[5] -> 6
[6] -> 7
[7] -> 8
[Prod1 > File] 9
[4] -> 5
[5] -> 6
[6] -> 7
[7] -> 8
[0] -> 9
[File > Cons2] 5
[5] -> 6
[6] -> 7
[7] -> 8
[0] -> 9
[Prod1 > File] 10
[5] -> 6
[6] -> 7
[7] -> 8
[0] -> 9
[1] -> 10
[Prod1 > File] 11
[5] -> 6
[6] -> 7
[7] -> 8
```

```
[0] -> 9
[1] -> 10
[2] -> 11
[File > Cons2] 6
[6] -> 7
[7] -> 8
[0] -> 9
[1] -> 10
[2] -> 11
[Prod1 > File] 12
[6] -> 7
[7] -> 8
[0] -> 9
[1] -> 10
[2] -> 11
[3] -> 12
[Prod1 > File] 13
[6] -> 7
[7] -> 8
[0] -> 9
[1] -> 10
[2] -> 11
[3] -> 12
[4] -> 13
[File > Cons2] 7
[7] -> 8
[0] -> 9
[1] -> 10
[2] -> 11
[3] -> 12
[4] -> 13
[Prod1 > File] 14
[7] -> 8
[0] -> 9
[1] -> 10
[2] -> 11
[3] -> 12
[4] -> 13
[5] -> 14
[Prod1 > File] 15
Prod1 s'endort
[File > Cons2] 8
[0] -> 9
[1] -> 10
[2] -> 11
[3] -> 12
[4] -> 13
[5] -> 14
[6] -> 15
Cons2 reveille prod
Prod1 se reveille
[Prod1 > File] 16
```

```
Prod1 s'endort
Prod1 se reveille
[File > Cons2] 9
[1] -> 10
[2] -> 11
[[Prod1 > File] 17
Prod1 s'endort
3] -> 12
[4] -> 13
[5] -> 14
[6] -> 15
[7] -> 16
[0] -> 17
[File > Cons2] 10
[2] -> 11
[3] -> 12
[4] -> 13
[5] -> 14
[6] -> 15
[7] -> 16
[0] -> 17
Cons2 reveille prod
Prod1 se reveille
[Prod1 > File] 18
Prod1 s'endort
Prod1 se reveille
[File > Cons2] 11
[3] -> 12
[4] -> 13
[5] -> 14
[6] -> 15
[7] -> 16
[0] -> 17
[1] -> 1[Prod1 > File] 19
Prod1 s'endort
8
[2] -> 19
[File > Cons2] 12
[4] -> 13
[5] -> 14
[6] -> 15
[7] -> 16
[0] -> 17
[1] -> 18
[2] -> 19
Cons2 reveille prod
Prod1 se reveille
------> FIN Prod1
[File > Cons2] 13
[5] -> 14
[6] -> 15
[7] -> 16
```

```
[0] -> 17
[1] -> 18
[2] -> 19
[File > Cons2] 14
[6] -> 15
[7] -> 16
[0] -> 17
[1] -> 18
[2] -> 19
[File > Cons2] 15
[7] -> 16
[0] -> 17
[1] -> 18
[2] -> 19
[File > Cons2] 16
[0] -> 17
[1] -> 18
[2] -> 19
[File > Cons2] 17
[1] -> 18
[2] -> 19
[File > Cons2] 18
[2] -> 19
[File > Cons2] 19
Fifo vide
Cons2 s'endort
Erreur Cons2 : la file est vide
Fifo vide
Cons2 s'endort
Plus rien a ordonnancer.
Sortie du noyau

Activations tache 0 : 3
Activations tache 1 : 47
Activations tache 2 : 99
Activations tache 3 : 0
Activations tache 4 : 0
Activations tache 5 : 0
Activations tache 6 : 0
Activations tache 7 : 0
```

Les résultats mettent en évidence qu'aucun accès concurrent n'est permis, nous avons bien la production et la consommation qui se font sans erreur d'accès sur la file partagée. Vérifions ceci avec deux fois plus de chaque tâche.

Avec 2 producteurs et 2 consommateurs :

Finalement ce dernier exemple nous permet de valider la bonne construction et utilisation de nos sémaphores. On peut observer les différents consommateurs et producteurs accèder et modifier à la file partagée sans que celle-ci n'est de valeur erronée.

```
Test noyau Semaphore
------> EXEC tache A
```

```
------> DEBUT Prod1
------> DEBUT Prod2
------> DEBUT Cons3
[Prod1 > File] 1
[0] -> 1
Prod1 reveille cons
------> DEBUT Cons4
[Prod2 > File] 2
[0] -> 1
[1] -> 2
[Prod1 > File] 3
[0] -> 1
[1] -> 2
[2] -> 3
[Prod2 > File] 4
[0] -> 1
[1] -> 2
[2] -> 3
[3] -> 4
[File > Cons3] 1
[1] -> 2
[2] -> 3
[3] -> 4
[File > Cons4] 2
[2] -> 3
[3] -> 4
[Prod2 > File] 6
[2] -> 3
[3] -> 4
[4] -> 6
[Prod1 > File] 5
[2] -> 3
[3] -> 4
[4] -> 6
[5] -> 5
[Prod1 > File] 7
[2] -> 3
[3] -> 4
[4] -> 6
[5] -> 5
[6] -> 7
[File > Cons3] 3
[3] -> 4
[4] -> 6
[5] -> 5
[6] -> 7
[Prod2 > File] 8
[3] -> 4
[4] -> 6
[5] -> 5
[6] -> 7
[7] -> 8
```

```
[File > Cons4] 4
[4] -> 6
[5] -> 5
[6] -> 7
[7] -> 8
[Prod2 > File] 10
[4] -> 6
[5] -> 5
[6] -> 7
[7] -> 8
[0] -> 10
[Prod1 > File] 9
[4] -> 6
[5] -> 5
[6] -> 7
[7] -> 8
[0] -> 10
[1] -> 9
[Prod1 > File] 11
[4] -> 6
[5] -> 5
[6] -> 7
[7] -> 8
[0] -> 10
[1] -> 9
[2] -> 11
[File > Cons3] 6
[5] -> 5
[6] -> 7
[7] -> 8
[0] -> 10
[1] -> 9
[2] -> 11
[Prod2 > File] 12
[5] -> 5
[6] -> 7
[7] -> 8
[0] -> 10
[1] -> 9
[2] -> 11
[3] -> 12
[File > Cons4] 5
[6] -> 7
[7] -> 8
[0] -> 10
[1] -> 9
[2] -> 11
[3] -> 12
[Prod2 > File] 14
[6] -> 7
[7] -> 8
[0] -> 10
```

```
[1] -> 9
[2] -> 11
[3] -> 12
[4] -> 14
[Prod1 > File] 13
Prod1 s'endort
[6] -> 7
[7] -> 8
[1] -> 9
[2] -> 11
[3] -> 12
[4] -> 14
[5] -> 13
Prod2 s'endort
[File > Cons3] 7
[7] -> 8
[0] -> 10
[1] -> 9
[2] -> 11
[3] -> 12
[4] -> 14
[5] -> 13
Cons3 reveille prod
Prod1 se reveille
[File > Cons4] 8
[0] -> 10
[1] -> 9
[2] -> 11
[3] -> 12
[4] -> 14
[5] -> 13
Prod2 se reveille
[Prod1 > File] 15
[0] -> 10
[1] -> 9
[2] -> 11
[3] -> 12
[4] -> 14
[5] -> 13
[6] -> 15
[Prod2 > File] 16
Prod2 s'endort
[File > Cons3] 10
[1] -> 9
[2] -> 11
[3] -> 12
[4] -> 14
[5] -> 13
[6] -> 15
[7] -> 16
Cons3 reveille prod
[Prod1 > File] 17
```

```
Prod1 s'endort
[File > Cons4] 9
[2] -> 11
[3] -> 12
[4] -> 14
[5] -> 13
[6] -> 15
[7] -> 16
[0] -> 17
Cons4 reveille prod
Prod2 se reveille
Prod1 se reveille
[Prod2 > File] 18
Prod2 s'endort
Prod2 se reveille
------> FIN Prod2
Erreur Prod1 : la file est pleine !
Prod1 s'endort
[File > Cons3] 11
Prod1 se reveille
[3] -> 12
[4] -> 14
[5] -> 13
[6] -> 15
[7] -> 16
[0] -> 17
[1] -> 18
[File > Cons4] 12
[4] -> 14
[5] -> 13
[6] -> 15
[7] -> 16
[0] -> 17
[1] -> 18
[Prod1 > File] 19
[4] -> 14
[5] -> 13
[6] -> 15
[7] -> 16
[0] -> 17
[1] -> 18
[2] -> 19
Cons3 reveille prod
Cons4 reveille prod
------> FIN Prod1
[File > Cons3] 14
[5] -> 13
[6] -> 15
[7] -> 16
[0] -> 17
[1] -> 18
[2] -> 19
```

```
[File > Cons4] 13
[6] -> 15
[7] -> 16
[0] -> 17
[1] -> 18
[2] -> 19
[File > Cons3] 15
[7] -> 16
[0] -> 17
[1] -> 18
[2] -> 19
[File > Cons4] 16
[0] -> 17
[1] -> 18
[2] -> 19
[File > Cons4] 17
[1] -> 18
[2] -> 19
[File > Cons3] 18
[2] -> 19
[File > Cons3] 19
Fifo vide
Cons3 s'endort
Erreur Cons4 : la file est vide
Fifo vide
Cons4 s'endort
Erreur Cons3 : la file est vide
Fifo vide
Cons3 s'endort
Plus rien a ordonnancer.
Sortie du noyau

Activations tache 0 : 5
Activations tache 1 : 30
Activations tache 2 : 24
Activations tache 3 : 55
Activations tache 4 : 50
Activations tache 5 : 0
Activations tache 6 : 0
```