Université de Technologie de Compiègne

MI11

Rapport des TPs : Réalisation d'un mini noyau temps réel

Clément BLANQUET et Rafik CHENNOUF

Juin 2017

Sommaire

Table des figures

Chapitre 1

Rapport TP 2 - Partie 3 et 4

Le but de ce TP est d'implémenter des fonctions d'exclusions mutuelles afin que plusieurs tâches ne puissent pas accéder à une section critique en même temps au risque de créer un interblocage.

Exercice 1 : Exclusion mutuelle

Tout d'abord, il est possible de faire du partage de ressources en agissant directement sur les tâches dépendantes en les faisant s'endormir ou se réveiller selon la situation. Dès qu'une tâche a terminée son accès à la mémoire partagée elle s'endort et réveille l'autre tâche afin qu'elle puisse y avoir accès et vis-versa.

1.1 Suspension d'une tâche

L'endormissement d'une tâche se fait via la primitive *dort()* du fichier **noyau.c**. Le but de cette fonction est de suspendre la tâche courante qui passe donc de l'état **EXEC** pour 'exécuter' à l'état **SUSP** pour 'suspendre'. La tâche est ensuite retirée de la file ds tâches et un appel à l'ordonnanceur est réalisé afin de charger la tâche suivante. De plus, toutes ces opérations constituent une section critique qui ne doivent pas être exécutées en même temps par plusieurs fonctions. C'est pour cela qu'il faut les protéger avec un *mutex* ou un *lock*.

Ci-dessous le code de la fonction *dort()* :

```
void  dort(void)
{
  _lock_();        // section critique

  CONTEXTE *p = &_contexte[_tache_c];
  p->status = SUSP; // suspension

  retire(_tache_c); // retirer la tâche
  schedule();

  _unlock_();
}
```

1.2 Réveil d'une tâche

Le réveil d'une tâche se fait via la primitive *reveille()* du fichier **noyau.c**. Cette primitive fonctionne de la même manière que la fonction *dort()* vue précédemment sauf que l'état de la tâche courante est passé en mode EXEC au lieu de SUSP afin que la tâche puisse être exécutable par l'ordonnanceur après l'avoir ajoutée dans la file.

Ci-dessous le code de la fonction *reveille()* :

```
void reveille(uint16_t t)
{

  // on vérifie que la tâche existe et est suspendue
  if(t > MAX_TACHES || _contexte[t].status != SUSP)
    return;

  _lock_();    // section critique

  CONTEXTE *p = &_contexte[t];
  p->status = EXEC;    // exécution

  ajoute(t);     // ajout dans la file
  schedule();

  _unlock_();
}
```

1.3 Modèle de communications producteur/consommateur.

Afin de tester nos deux primitives *dort()* et *reveille()*, nous avons implémenté le modèle de communications producteur/consommateur. Tout d'abord le programme comporte deux tâches ; la première, le producteur, produit des entiers dans une file circulaire, la seconde, le consommateur, retire ces entiers de la file et les affiche.

Pour faire cela, nous disposons d'une FIFO sous forme d'un tableau d'entiers de taille fixée. On distingue 4 cas possibles :

1) Le producteur a tellement produit que la file est pleine => il s'endort. 2) Le producteur a produit au moins un entier, la file est non vide => il réveille le consommateur pour qu'il consomme un ou des entiers. 3) La file est vide car le producteur n'y a rien produit => le consommateur s'endort. 4) Il reste encore de

la place dans la file, la file est non pleine => le producteur se réveille pour produire des entiers.

Pour gérer tous ces cas, nous possédons une variable qui compte le nombre de places libres et qui est, au début du programme, initialisée à la taille du tableau. Lorsque le nombre de places libres est supérieur ou égal à 1, le producteur produit un entier dans la file puis décrémente le nombre de places libres. De même, si le nombre de places libres est inférieur à la taille totale de la file alors le consommateur consomme un entier puis incrémente le nombre de places libres.
Le cas 1 se produit lorsque la file est pleine, c'est à dire lorsque le nombre de places libres est égal à 0. Le cas 2 se produit lorsque la file est non vide, c'est à dire lorsque le nombre de places libres est inférieur à la taille totale de la file. Le cas 3 se produit lorsque la file est vide, c'est à dire lorsque le nombre de places libres est égal à la taille totale de la file. Le cas 4 se produit lorsque la file est non pleine, c'est à dire lorsque le nombre de places libres est égal à 1.

En résumé, le producteur produit des entiers dans la file tant que celle-ci n'est pas pleine sinon il s'endort et le consommateur lit ces entiers tant que la file n'est pas vide sinon il s'endort. Lorsqu'il y a la moindre place dans la file, le consommateur réveille le producteur et lorsqu'il y a le moindre entier dans la file, le producteur réveille le consommateur. A noté aussi que l'accès à la file représente une zone critique qu'il faut protéger via un *lock*. Initialement, seul le producteur est réveillé et le consommateur est endormi car il faut pouvoir produire au moins un entier.

Nous avons testé deux cas de figures, un cas où le producteur est plus rapide que le consommateur et le cas inverse.

Ci-dessous le code du modèle de communications producteur/consommateur dans le cas où le consommateur est plus rapide que le producteur :

```
1  #define TAILLE_TABLEAU 3

3  TACHE tacheStart();
   TACHE tacheProd();
5  TACHE tacheConso();
   uint16_t prod, conso;
7  uint16_t fifo[TAILLE_TABLEAU];  // la file
   uint16_t nb_places_libres = TAILLE_TABLEAU;
9
   TACHE tacheStart(void)     // tâche de démarrage
11 {
     puts("------> EXEC tache Start");
13   prod = cree(tacheProd);
     conso = cree(tacheConso);
```

```
15
   active(prod);
17 active(conso);

19 fin_tache();
}
21
TACHE tacheProd(void)    // tâche producteur
23 {
   puts("------> EXEC tache Prod");
25
   uint16_t j=0,k,i=0;
27
   while(1)
29 {
     for (k=0; k<30000; k++);     // producteur plus lent
31
     if (nb_places_libres>=1)     // production d'un entier
33   {
       _lock_();                  // zone critique
35     printf("**PROD** -> Production : fifo[%d] = %d\n", i, j);
       fifo[i]=j;
37     _unlock_();
       j++;
39
       nb_places_libres--;
41
       i++;
43     if (i==TAILLE_TABLEAU)  // i = indice de la file
         i=0;
45   }
     else      // file pleine
47   {
       _lock_();
49     puts("**PROD** -> Producteur dort (Cas 1 : File pleine)\n");
       _unlock_();
51
       dort();   // producteur s'endort
53   }

55   if (nb_places_libres<TAILLE_TABLEAU)  // file non vide
     {
57     _lock_();
       puts("**PROD** -> Reveil du conso (Cas 2 : File non vide)\n");
59     _unlock_();

61     reveille(conso);     // consommateur se réveille
     }
63
```

```
  }

  fin_tache();
}

TACHE tacheConso(void)     // tâche consommateur
{
  uint16_t k,i=0;

  puts("------> EXEC tache Conso");

  dort();   // consommateur dort initialement

  while(1)
  {
    for (k=0; k<10000; k++);  // consommateur plus rapide

    if (nb_places_libres==TAILLE_TABLEAU) // file vide
    {
      _lock_();
      puts("**CONSO** -> Consommateur dort (Cas 3 : File vide)\n");
      _unlock_();

      dort();   // consommateur s'endort
    }
    else
    {
      _lock_();
      printf("**CONSO** -> Lecture de fifo[%d] = %d\n", i, fifo[i]);
      _unlock_();

      nb_places_libres++;

      i++;
      if (i==TAILLE_TABLEAU)  // i = indice de la file
        i=0;
    }

    if (nb_places_libres==1)  // file non pleine
    {
      _lock_();
      puts("**CONSO** -> Reveil prod (Cas 4 : File non pleine)\n");
      _unlock_();

      reveille(prod);   // producteur se réveille
    }
  }

  fin_tache();
```

```
113 }

115 int main()
{
117   serial_init(115200);
   puts("Test noyau");
119   puts("Noyau preemptif");
   puts("********************DEBUT**********************\n\n\n\n\n");
121
   printf("TAILLE DU TABLEAU : %d\n", TAILLE_TABLEAU);
123
   start(tacheStart);
125   return(0);
}
```

Le résultat obtenu lorsque le consommateur est plus rapide que le producteur est donné sur la figure 1.3.

Initialement, le producteur écrit l'entier 0 à la case 0 de la file puis il réveille le consommateur (Cas 2 : file non vide). Le consommateur qui est très rapide s'empresse aussitôt de lire l'entier 0 à la case 0 puis s'endort (Cas 3 : file vide). Ensuite, le producteur écrit l'entier 1 à la case 1 de la file puis il réveille le consommateur pour qu'il lise cet entier.
Comme le consommateur est plus rapide que le producteur, nous obtenu une succession d'écriture/lecture. Dès qu'un entier est présent dans la file, le consommateur le lit. Cliquez sur l'animation suivante pour voir le résultat de notre programme (nécessite de lire ce PDF avec Adobe Acrobat Reader) :

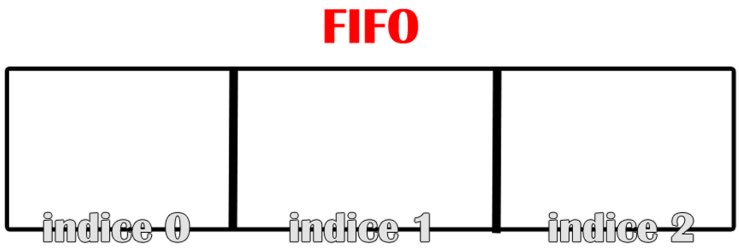

FIGURE 1.1 – Consommateur plus rapide que le producteur

Nous pouvons aussi tester le cas où le producteur est plus rapide que le consommateur (figure 1.4).

Dans ce cas, le producteur enchaîne deux écritures à la suite (entier 0 à la case 0 et entier 1 à la case 1) car il est plus rapide que le consommateur qui n'a pas le temps de lire le premier entier. Après cela, le consommateur peut enfin lire l'entier 0 à la case 0 puis le producteur reprend la main et enchaîne de nouveau deux écritures (entier 2 à la case 2 et entier 3 à la case 0). Le consommateur se réveille et lit donc l'entier 1 à la case 1, on se retrouve donc dans le cas 4 (file non pleine). Le producteur reprend la main et écrit l'entier 4 à la case 1 qui vient d'être libérée par le consommateur puis il s'endort (Cas 1 : file pleine). Le producteur aura toujours une longueur d'avance sur le consommateur et sera souvent endormi à cause d'une file pleine. Cliquez sur l'animation suivante pour voir le résultat de notre programme (nécessite de lire ce PDF avec Adobe Acrobat Reader) :

FIGURE 1.2 – Producteur plus rapide que le consommateur

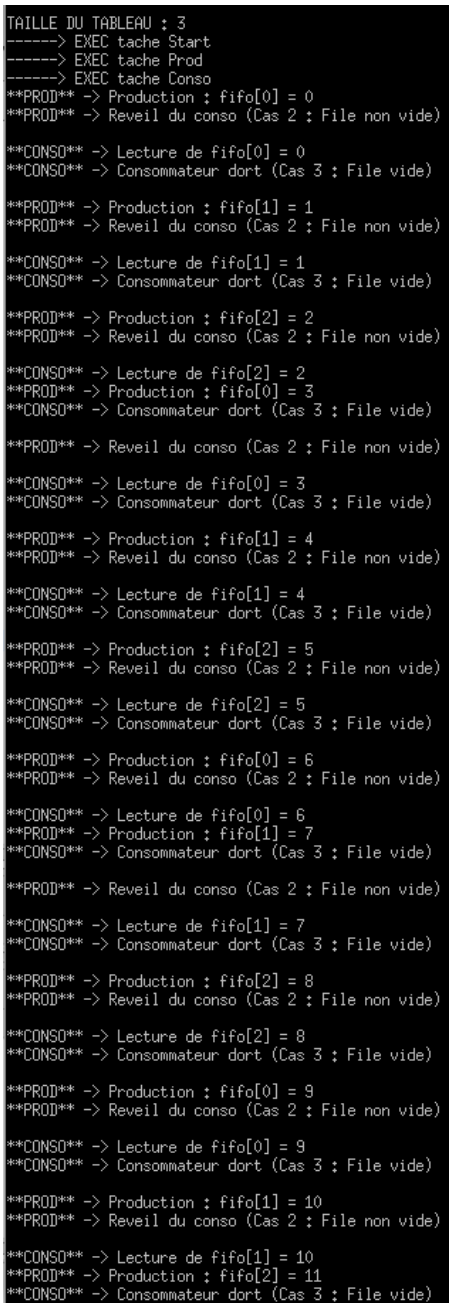

FIGURE 1.3 – Consommateur plus rapide que le producteur

FIGURE 1.4 – Producteur plus rapide que le consommateur