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 1 - Partie 1 et 2

1ère partie : Ordonnanceur de tâches

Voici comment nous avons codé les fonctions *file_init()*, *ajoute(n)*, *suivant()* et *retire(n)* :

1.1 file_init()

Cette fonction permet simplement d'initialiser la file FIFO de tâches.

```
void  file_init( void )
{
  int i;
  _queue=F_VIDE;

  for (i=0;i<MAX_TACHES;i++)
    _file[i]=F_VIDE;
}
```

Dans cette fonction, on se contente d'initialiser la queue da la file FIFO à la valeur F_VIDE (F_VIDE = MAX_TACHES = 8), ce qui signifie que la file est vide. De plus, on initialise chaque case du tableau représentant la file à la même valeur, indiquant que chaque case est vide.

1.2 ajoute(n)

Cette fonction permet d'ajouter une tâche en fin de file.

```
void  ajoute ( uint16_t n )
{
  if(n > MAX_TACHES - 1) // Si le numéro de la tâche est trop grand
    return;

  if(_queue == F_VIDE) // Si la file est vide
  {
    _queue = n;
    _file[_queue] = n;
  }
```

```
  else // La file n'est pas vide
  {
    uint16_t tmp = _file[_queue];
    _file[_queue] = n;
    _file[n] = tmp;
    _queue = n;
  }
}
```

La fonction *ajoute* prend en argument un entier qui représente le numéro de la tâche à ajouter.

On commence par regarder si le numéro de la tâche à ajouter est cohérent (inférieur ou égal à 7). Si ce n'est pas le cas, on sort de la fonction.

Sinon, si la file est vide (_queue = F_vide), la queue devient la tâche qui vient d'être ajoutée et le successeur de cette tâche est elle-même.

Enfin, si la file n'est pas vide, on veut ajouter la nouvelle tâche en fin de file. Autrement dit, la queue doit valeur cette même nouvelle tâche. On commence par sauvegarder dans un tampon le successeur de la queue actuelle. Ensuite, on désigne la nouvelle tâche comme étant le successeur de la queue actuelle. De plus, on désigne le successeur de la nouvelle tâche comme étant la tâche sauvegardée dans le tampon, anciennement successeur de la queue actuelle. Enfin, on désigne notre nouvelle tâche comme étant la queue.

De cette façon, la nouvelle tâche s'est intercalée entre l'ancienne queue et son ancien successeur, et est devenue la nouvelle queue.

1.3 suivant()

Cette fonction retourne la tâche à activer, et met à jour _queue pour qu'elle pointe sur la suivante.

```
uint16_t  suivant( void )
{
  _queue = _file[_queue];
  return _queue;
}
```

Ici on assigne à la queue son successeur (donc la tâche à activer) puis on la retourne.

1.4 retire(n)

Cette fonction permet de retirer une tâche de la file sans en modifier l'ordre.

```
1  void  retire( uint16_t t )
   {
3    int i = 0;

5    if(t > MAX_TACHES || _queue == F_VIDE)
       return;
7
     while(_file[i] != t)
9      i++;          // i = prédecesseur de t
                     // ex : ( 1 -> 0 -> 2 ) => avec t = 0, i = 1
11
     if(i == MAX_TACHES)
13     return;

15   _file[i] = _file[t];   // 1 -> 2
     _file[t] = F_VIDE;
17   //_queue = _file[i];

19   if(t == _queue)
     {
21
       if(t == i)       // si le prédecesseur est lui même => une seule
      tâche => queue => on vide
23       _queue = F_VIDE;
       else
25       _queue = i;    // la queue devient le prédecesseur de la tâche
      qu'on retire
     }
27 }
```

La fonction prend en argument le numéro de la tâche à retirer. On commence donc par vérifier que ce numéro est cohérent (inférieur ou égal à 7) et que la file n'est pas vide auquel cas il est impossible de retirer un élément.

Une fois cette vérification effectuée, on cherche le prédecesseur de la tâche à retirer. Si on ne trouve rien, cela signifie que la tâche n'existe pas et on sort donc de la fonction.

Sinon, on "saute au dessus" de la tâche à retirer, c'est à dire qu'on fixe le successeur du prédécesseur de cette tâche comme étant le successeur de cette même tâche.

Enfin, on doit vérifier si la tâche retirée était la queue ou non.

— Si oui

- Si la file ne contient qu'un élément (donc le prédécesseur de la tâche est la tâche elle-même), on assigne à _queue la valeur F_VIDE (car la file est alors vide)
- Sinon, la queue devient le prédécesseur de la tâche que l'on a retiré.

— Sinon, ne rien faire de plus.

1.5 affic_file()

Cette fonction permet d'afficher la file.

```
1  void affic_file( void )
   {
3    int i = 0;

5    while (_file[i] == F_VIDE) // Recherche du premier élément de la
       file
       i++;
7
     int temp = i; // temp ==> premier élément de la file
9
     while (_file[i] !=temp) // Tant qu'on a pas fait tout le tour de la
        file
11   {
       if (i==_queue)
13       printf(" %d(Q) -> ", i); // Si l'élément i est la queue,
       afficher un "(Q)" à côté
       else
15         printf(" %d -> ", i); // Sinon l'afficher normalement
       i=_file[i]; // Passer à l'élément suivant
17   }
     // Affichage du dernier élément
19   if (i==_queue)
       printf(" %d (Q) -> ", i);
21   else
       printf(" %d -> ", i);
23
     printf("\n");
25 }
```

On commence par chercher le premier élément de la file. Ensuite, on effectue une boucle d'affichage tant qu'un tour complet de la file n'a pas été effectué. On affiche "(Q)" à côté de la tâche sur laquelle la queue pointe.

1.6 Test de ces fonctions

Nous avons pu tester le bon fonctionnement de ces fonctions grâce au fichier de test *testfile.c* suivant :

```c
#include "noyau.h"
#include "serialio.h"


int main()
{
  serial_init(115200);

  printf("**********************************\n");

  file_init();

  // Ajouts des différentes tâches comme dans l'exemple du sujet
  ajoute(3);
  ajoute(5);
  ajoute(1);
  ajoute(0);
  ajoute(2);

  affic_file(); // Affichage de la file

  affic_queue(); // Affichage de la queue
  printf("Suivant() \n");
  suivant(); // On fait un "suivant()"
  affic_queue(); // On vérifie la queue après le "suivant()"

  affic_file(); // Affichage de la file avec la nouvelle queue

  printf("Retire 0 : \n");
  retire(0); // On retire la tâche 0
  affic_file(); // Affichage de la file après le "retire(0)"
  printf("Ajoute 6 : \n");
  ajoute(6); // On ajoute 6
  affic_file(); // Affichage de la file après le "ajoute(6)"

  return 0;
}
```

A l'exécution, on obtient :

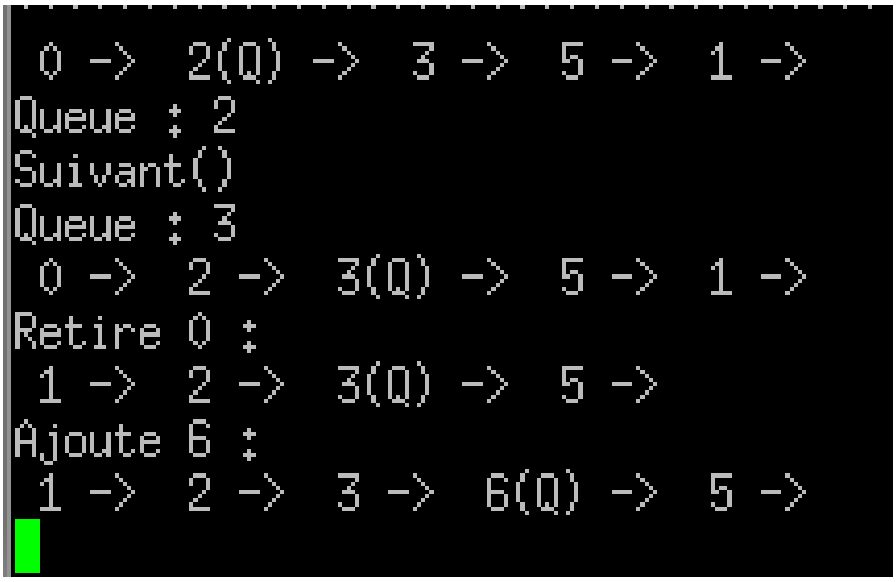

FIGURE 1.1 – Résultat de testfile.c

C'est bien le résultat attendu. En effet, au départ la queue vaut 2 Car c'est la dernière tâche à avoir été ajoutée. Après le suivant, la valeur de la queue change et devient le successeur de 2 à savoir 3, ce qui s'affiche correctement. Le *retire(0)* fonctionne bien aussi puisque la tâche 0 a disparu de la file et les autres tâches ont leurs successeurs et prédécesseurs bien mis à jour. Enfin, le ajoute(6) ajoute bien la tâche 6 en fin de file en mettant à jour la queue.

2ème partie : gestion et commutation de tâches

Dans cette seconde partie, nous devons réaliser les primitives de gestion des tâches du mini noyau temps réel, ainsi que le système de commutation de tâches. Tout cela est écrit dans le fichier *noyau.c*.

Les codes nous étant fournis, nous devons les expliquer. Notons tout d'abord les variables internes au noyau :

```
static int compteurs[MAX_TACHES]; // Compteurs d'activations
volatile uint16_t _tache_c;       // numéro de tache courante
uint32_t  _tos;                   // adresse du sommet de pile
int  _ack_timer = 1;           // = 1 si il faut acquitter le timer
```

2.1 Fonction noyau_exit()

```
void  noyau_exit(void)
{
  int j;
  _irq_disable_();                // Désactiver les interruptions
  printf("Sortie du noyau\n");
  for (j=0; j < MAX_TACHES; j++)
    printf("\nActivations tache %d : %d", j, compteurs[j]);
  for(;;);                        // Terminer l'exécution
}
```

Commençons par la fonction *noyau_exit()* qui permet de sortir du noyau (donc d'arrêter tout). Les interruptions sont désactivées et juste avant de sortir du noyau (en faisant une boucle infinie), on affiche le nombre de fois que chaque tâche a été activée grâce au tableau *compteurs*.

2.2 Fonction fin_tache()

```
void  fin_tache(void)
{
  // on interdit les interruptions
  _irq_disable_();
  // la tache est enlevee de la file des taches
  _contexte[_tache_c].status = CREE;
  retire(_tache_c);
  schedule();
}
```

Cette fonction permet de rendre une tâche inactive alors qu'elle était active jusqu'alors. Pour cela, après avoir interdit les interruptions, on change son statut à "CREE" ce qui fait que la tâche n'est plus prête à être exécutée. Elle reste tout de même connue du noyau et est simplement en attente. On la retire de la FIFO grâce à notre fonction *retire()* créée dans la première partie. A la fin, on lance la fonction *schedule()* (que l'on étudiera par la suite) pour que les tâches suivantes soient puissent s'exécuter. En fait, on appelle cette fonction à la fin de chaque tâche pour y mettre fin.

2.3 Fonction cree(TACHE_ADR adr_tache)

```
uint16_t cree( TACHE_ADR adr_tache )
{
  CONTEXTE *p;                      // pointeur d'une case de _contexte
  static    uint16_t tache = -1;    // contient numero dernier cree


  _lock_();                         // debut section critique
  tache++;                          // numero de tache suivant

  if (tache >= MAX_TACHES)          // sortie si depassement
    noyau_exit();

  p = &_contexte[tache];            // contexte de la nouvelle tache

  p->sp_ini = _tos;                 // allocation d'une pile a la tache
  _tos -= PILE_TACHE + PILE_IRQ;    // decrementation du pointeur de
    pile pour la prochaine tache

  _unlock_();                       // fin section critique

  p->tache_adr = adr_tache;         // memorisation adresse debut de
    tache
  p->status = CREE;                 // mise a l'etat CREE
  return(tache);                    // tache est un uint16_t
}
```

La fonction *cree* permet de créer une tâche et de lui allouer une pile et un numéro.

2.4 Fonction active(uint16_t tache)

```
void  active( uint16_t tache )
{
  CONTEXTE *p = &_contexte[tache]; // acces au contexte tache

  if (p->status == NCREE)
    noyau_exit();                  // sortie du noyau

  _lock_();                        // debut section critique
  if (p->status == CREE)           // n'active que si receptif
  {
    p->status = PRET;              // changement d'etat, mise a l'etat
    PRET
    ajoute(tache);                 // ajouter la tache dans la liste
    schedule();                  // activation d'une tache prete
  }
  _unlock_();                      // fin section critique
}
```

Cette fonction place une tâche dans la file d'attente des tâches éligibles. Elle prend en entrée le numéro de la tâche.

On vérifie d'abord si la tâche est bien créée. Si c'est le cas, alors on va modifier le statut de la tâche (de CREE à PRET) et l'ajouter à notre FIFO (on rentre donc dans une section critique). On finit par lancer un *schedule()* pour activer la prochaine tâche.

2.5 Fonction scheduler

```
void __attribute__((naked)) scheduler( void )
{
  register CONTEXTE *p;
  register unsigned int sp asm("sp");  // Pointeur de pile

  // Sauvegarder le contexte complet sur la pile IRQ
  __asm__ __volatile__(
      "stmfd  sp, {r0-r14}^\t\n"  // Sauvegarde registres mode system
      "nop\t\n"                   // Attendre un cycle
      "sub    sp, sp, #60\t\n"    // Ajustement pointeur de pile
      "mrs    r0, spsr\t\n"       // Sauvegarde de spsr_irq
      "stmfd  sp!, {r0, lr}\t\n");// et de lr_irq

  if (_ack_timer)                 // Réinitialiser le timer si
    nécessaire
  {
```

```
16      register struct imx_timer* tim1 = (struct imx_timer *)
        TIMER1_BASE;
        tim1->tstat &=~TSTAT_COMP;
18    }
      else
20    {
        _ack_timer = 1;
22    }

24    _contexte[_tache_c].sp_irq = sp;// memoriser le pointeur de pile
      _tache_c = suivant();           // recherche du suivant
26    if (_tache_c == F_VIDE)
      {
28      printf("Plus rien à ordonnancer.\n");
        noyau_exit();                 // Sortie du noyau
30    }
      compteurs[_tache_c]++;          // Incrémenter le compteur d'
        activations
32    p = &_contexte[_tache_c];       // p pointe sur la nouvelle tache
        courante

34    if (p->status == PRET)          // tache prete ?
      {
36      sp = p->sp_ini;               // Charger sp_irq initial
        _set_arm_mode_(ARMMODE_SYS);  // Passer en mode système
38      sp = p->sp_ini - PILE_IRQ;    // Charger sp_sys initial
        p->status = EXEC;             // status tache -> execution
40      _irq_enable_();               // autoriser les interuptions
        (*p->tache_adr)();            // lancement de la tâche
42    }
      else
44    {
        sp = p->sp_irq;               // tache deja en execution,
        restaurer sp_irq
46    }

48    // Restaurer le contexte complet depuis la pile IRQ
      __asm__ __volatile__(
50        "ldmfd  sp!, {r0, lr}\t\n"  // Restaurer lr_irq
          "msr    spsr, r0\t\n"       // et spsr_irq
52        "ldmfd  sp, {r0-r14}^\t\n"  // Restaurer registres mode system
          "nop\t\n"                   // Attendre un cycle
54        "add    sp, sp, #60\t\n"    // Ajuster pointeur de pile irq
          "subs   pc, lr, #4\t\n");   // Retour d'exception
56  }
```

A EXPLIQUER A EXPLIQUER A EXPLIQUER A EXPLIQUER A EXPLI-

QUER A EXPLIQUER

2.6 Fonction schedule()

```
void  schedule( void )
{
  _lock_();                          // Debut section critique

  // On simule une exception irq pour forcer un appel correct à
    scheduler().
  _ack_timer = 0;
  _set_arm_mode_(ARMMODE_IRQ);       // Passer en mode IRQ
  __asm__ __volatile__(
      "mrs  r0, cpsr\t\n"            // Sauvegarder cpsr dans spsr
      "msr  spsr, r0\t\n"
      "add  lr, pc, #4\t\n"          // Sauvegarder pc dans lr et
                                     // l'ajuster
      "b    scheduler\t\n"           // Saut à scheduler
      );
  _set_arm_mode_(ARMMODE_SYS);       // Repasser en mode system

  _unlock_();                        // Fin section critique
}
```

La fonction *schedule()* permet en fait de faire un appel à la fonction *scheduler()*. Dans un section critique, elle commence par passer en mode IRQ (car la fonction *scheduler* doit s'exécuter dans ce mode). Ensuite, comme on l'a vu en cours, lors d'un changement de tâche, on doit :

- Copier cpsr dans spsr_mode :

```
"mrs  r0, cpsr\t\n"
"msr  spsr, r0\t\n"
```

- Changer cpsr
  - Passage en mode d ?exception
  - Interdiction des IRQ / FIQ si nécessaire
- Sauver pc (r15) dans lr_mode (r14_mode)
- Charger l'adresse du vecteur dans pc
- A la fin du traitement :
  - Restaurer cpsr depuis spsr_mode
  - Restaurer pc depuis lr_mode

2.7 Fonction start(TACHE_ADR adr_tache)

```
void  start( TACHE_ADR adr_tache )
{
  short j;
  register unsigned int sp asm("sp");
  struct imx_timer* tim1 = (struct imx_timer *) TIMER1_BASE;
  struct imx_aitc* aitc = (struct imx_aitc *) AITC_BASE;

  for (j=0; j<MAX_TACHES; j++)
  {
    _contexte[j].status = NCREE;     // initialisation de l'etat des
    taches
  }
  _tache_c = 0;                      // initialisation de la tache
    courante
  file_init();                       // initialisation de la file

  _tos = sp;                         // Haut de la pile des tâches
  _set_arm_mode_(ARMMODE_IRQ);       // Passer en mode IRQ
  sp = _tos;                         // sp_irq initial
  _set_arm_mode_(ARMMODE_SYS);       // Repasser en mode SYS

  _irq_disable_();                   // on interdit les interruptions

  // Initialisation du timer à 100 Hz
  tim1->tcmp = 10000;
  tim1->tprer = 0;
  tim1->tctl |= TCTL_TEN | TCTL_IRQEN | TCTL_CLKSOURCE_PERCLK16;

  // Initialisation de l'AITC
  aitc->intennum = TIMER1_INT;

  active(cree(adr_tache));           // creation et activation
    premiere tache
}
```

Cette fonction permet de lancer la première tâche et donc de lancer le système.

On commence par initialiser les statuts des tâches à "NCREE", la FIFO et la tâche courante à 0.

Chapitre 2

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 2.3.

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) :

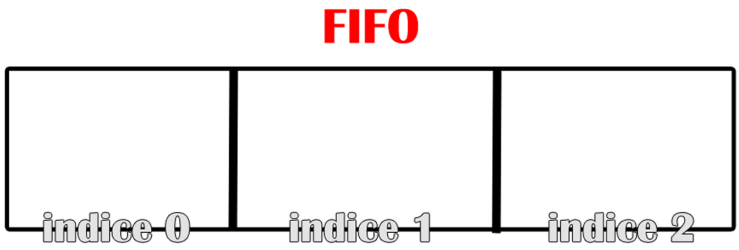

FIGURE 2.1 – Consommateur plus rapide que le producteur

Nous pouvons aussi tester le cas où le producteur est plus rapide que le consommateur (figure 2.4).

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 2.2 – Producteur plus rapide que le consommateur

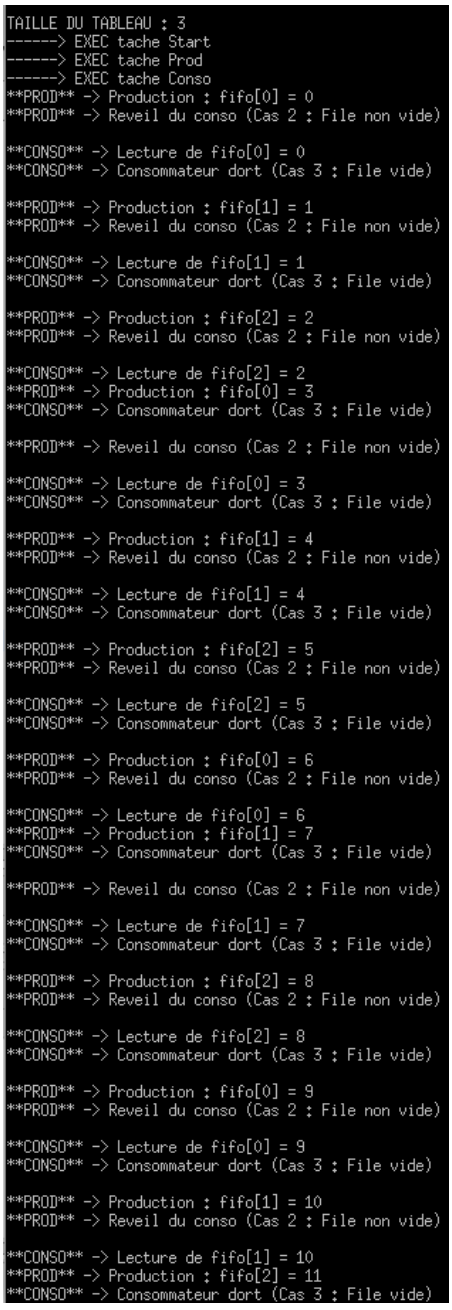

FIGURE 2.3 – Consommateur plus rapide que le producteur

FIGURE 2.4 – Producteur plus rapide que le consommateur

Chapitre 3

Rapport TP3 - Parties 5 et 6

5ème partie : le dîner des philosophes

6ème partie : communication par tubes

Dans cette partie nous devons réaliser un système de communication par tube. De chaque côté du tube ne se trouve qu'une seule et unique tâche.

Commençons par le fichier *pipe.h* :

```
#ifndef PIPE_H_
#define PIPE_H_

#include "noyau.h"

#define MAX_PIPES 5 //Nombre de tubes
#define SIZE_PIPE 10 //Taille de chaque tube

typedef struct
{
  ushort pr_w , pr_r ; // redacteur & lecteur du tube
  ushort ocupp ;  // donnees restantes
  uchar is, ie ;    // pointeurs d?entree / sortie
  uchar tube[SIZE_PIPE] ; // Tampon
} PIPE;

PIPE  _pipe[MAX_PIPES] ;   // Variables tubes


//Allocation du conduit
unsigned int p_open(unsigned int redacteur, unsigned int lecteur);
//Libération du tube
void p_close (unsigned int conduit);
//Lecture dans un tube
void p_read  (int tube, uchar* donnees, int quantite);
//Ecriture dans un tube
void p_write(int tube, uchar* donnees, int quantite);
//Initialisation des tubes
void init_pipes();
```

Nous avons choisi un nombre de tubes égal à 5 et une taille de 10 pour chaque tube.

2.1 Fonction init_pipes

```
void init_pipes()
{
  unsigned int i;
  for (i=0;i<MAX_PIPES;i++)
    _pipe[i].pr_w=MAX_TACHES; //Tube inutilisé
}
```

La fonction *init_pipes()* initialise tous les tubes au départ en leur assignant la constante MAX_TACHES comme écrivain, ce qui signifie qu'ils sont inutilisés.

2.2 Fonction p_open

```
//Ouvre un nouveau pipe
unsigned int p_open(unsigned int redacteur, unsigned int lecteur)
{
  //VérifIier si les tâches sont créées
  if (_contexte[redacteur].status == NCREE || _contexte[lecteur].
   status == NCREE)
    return -1;

  //Vérifier qu'il n'existe pas de tube avec ces 2 taches
  unsigned int i;
  for (i=0 ; i<MAX_PIPES ; i++)
  {
    if (_pipe[i].pr_w == redacteur && _pipe[i].pr_r == lecteur)
      //Il existe un tube avec ces 2 taches
      return -1;
  }

  //Trouver un tube non utilisé
  i=0;
  while ( (_pipe[i].pr_w != MAX_TACHES) && (i < MAX_PIPES) )
    i++;
  if (i == MAX_PIPES) //Aucun tube n'est libre
    return -2;

  //Initialisation du tube
  _pipe[i].pr_w = redacteur;
  _pipe[i].pr_r = lecteur;
  _pipe[i].is = _pipe[i].ie = 0;
```

```
28    _pipe[i].ocupp = 0; //Données restantes (0 au départ)

30
    //Retourner le numéro du tube créé
32    return i;
}
```

Cette fonction permet d'ouvrir un nouveau tube. Elle prend comme arguments la tâche lectrice et la tâche rédactrice.

On commence par vérifier si ces deux tâches existent, et qu'il n'existe pas déjà de tube entre ces deux tâches. Ensuite, on cherche un tube non utilisé dans le tableau de tube *_pipe*. Une fois trouvé, on attribue les valeurs adéquates aux champs pr_w, pr_r, is, ie (tous les deux à zéro) et occup (à zéro aussi) du tube.

2.3 Fonction p_close

```
1  //Ferme un pipe
   void p_close (unsigned int conduit)
3  {
     _pipe[conduit].pr_w = MAX_TACHES;
5  }
```

La fonction *p_close* permet de ferme un tube en le rendant inutilisé en attribuant la valeur MAX_TACHES à son écrivain.

2.4 Fonction p_read

```
1  //Lit un certain nombre de données sur un pipe
   void p_read  (int tube, uchar* donnees, int quantite)
3  {
     //Vérifier que le tube existe et que la tâche en est propriétaire
5    if ( (_pipe[tube].pr_r != _tache_c) )
     {
7      printf("Tache non autorisee a lire dans le pipe\n");
       return;
9    }

11   //Lire les données à partir du tampon
     int i = 0;
13   for (i=0 ; i < quantite ; i++)
     {
15     //Vérifier que le tampon n?est pas vide, sinon endormir la tache
```

```
    if (_pipe[tube].ocupp == 0)
    {
      printf("Tampon vide ==> endormissement de la tache %d\n", _pipe
    [tube].pr_r);
      dort();
    }

    donnees[i] = _pipe[tube].tube[_pipe[tube].is];
    _pipe[tube].is++;
    _pipe[tube].ocupp--;

    //Si le tube était plein et si
    //la tâche écrivain est suspendue sur une
    //écriture dans ce tube alors la réveiller
    if ( (_pipe[tube].ocupp+1 == SIZE_PIPE) && (_contexte[_pipe[tube
    ].pr_w].status == SUSP) )
      reveille(_pipe[tube].pr_w);
  }
}
```

La fonction *p_read* lit une certaine quantité de données sur un tube. Elle prend comme argument le tube en question, un tableau de données et la quantité à lire.

On commence une fois de plus par une vérification. On vérifie si le tube demandé existe et si la tâche qui veut y lire en a bien le droit (c'est à dire on vérifie si le champ pr_r du tube est bien la tâche courante).

Ensuite, on peut commencer à lire les données du tube. Sachant que nos tubes sont des tableaux de caractères et qu'on lit une certaine quantité, on fait une boucle allant de 0 à cette quantité pour lire caractère par caractère.

A chaque lecture du tube, on vérifie s'il est vide ou non. S'il l'est, on l'endort. Sinon, on peut procéder à la lecture du caractère (dà l'emplacement de sortie "is"). On incrément le champ *is* du tube et on décrémente la champ *ocupp* car on vient de lire un caractère.

Enfin, si le tube était plein et si la tâche écrivain est suspendue sur une écriture dans ce tube (c'est à dire si l'écrivain, lors d'une tentation d'écriture est tombé sur un tube plein) alors on la réveille. De cette façon, une écriture sur un tube plein ne posera pas de problème : le rédacteur attendra simplement (en s'endormant) que le lecteur lise une donnée dans le tube et le réveille.

2.5 Fonction p_write

```
//Ecrit un certain nombre de données sur un pipe
void p_write(int tube, uchar* donnees, int quantite)
{
```

```
4    //Vérifier que le tube existe et que la tâche en est propriétaire
     if ( (_pipe[tube].pr_w != _tache_c) )
6    {
       printf("Tache non autorisee a ecrire dans le pipe\n");
8      return;
     }
10
     //Copie des données dans le tube
12   int i;
     for (i=0 ; i < quantite ; i++)
14   {
       //Vérifier qu?il y a de la place dans le tampon, sinon endormir
      la tache
16     if (_pipe[tube].ocupp == SIZE_PIPE)
       {
18       printf("Plus de place dans le tampon ==> endormissement de la
      tache %d\n", _pipe[tube].pr_w);
         dort();
20     }

22     //Copie des données
       _pipe[tube].tube[_pipe[tube].ie] = donnees[i];
24     _pipe[tube].ie++;
       _pipe[tube].ocupp++;
26
       //Si le tube était vide et si
28     //la tache lectrice est suspendue sur
       //une lecture de ce tube alors la réveiller
30     if ( (_pipe[tube].ocupp-1 == 0) && (_contexte[_pipe[tube].pr_r].
      status == SUSP) )
         reveille(_pipe[tube].pr_r);
32   }

34 }
```

La fonction *p_write* écrit une certaine quantité de données sur un tube. Elle prend en argument le tube désiré, un tableau de données et une quantité de données à écrire.

Tout comme pour la lecture, on vérifie si le tube demandé existe et si la tâche qui veut y écrire en a bien le droit (c'est à dire on vérifie si le champ pr_w du tube est bien la tâche courante).

On peut ensuite commencer à écrire dans le tube. On procède de la même manière que la lecture c'est à dire caractère par caractère.

Si le tube est plein, on endort le rédacteur jusqu'à ce que le lecteur le réveille après avoir lu un caractère. Sinon, on peut copier le caractère dans le tube (à

l'emplacement d'entrée "ie"). On incrémente ie et ocupp comme il se doit.

Enfin, si le tube était vide et si la tache lectrice est suspendue sur une lecture de ce tube (c'est à dire si un lecteur, en tentant de lire, est tombé sur un tube vide) alors on la réveille. De cette façon, une lecture sur un tube vide ne posera pas de problème : le lecteur attendra simplement (en s'endormant) que le rédacteur écrive dans le tube et le réveille.