



UNIVERSITÉ DE TECHNOLOGIE DE
COMPIÈGNE

MI11

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

Clément BLANQUET et Rafik CHENNOUF

Juin 2017

Sommaire

1	Rapport TP 1 - Partie 1 et 2	4
	1ère partie : Ordonnanceur de tâches	4
1.1	file_init()	4
1.2	ajoute(n)	4
1.3	suiwant()	5
1.4	retire(n)	6
1.5	affic_file()	7
1.6	Test de ces fonctions	8
	2ème partie : gestion et commutation de tâches	10
2.1	Fonction noyau_exit()	10
2.2	Fonction fin_tache()	10
2.3	Fonction cree(TACHE_ADR adr_tache)	11
2.4	Fonction active(uint16_t tache)	12
2.5	Fonction scheduler	12
2.6	Fonction schedule()	14
2.7	Fonction start(TACHE_ADR adr_tache)	15
2	Rapport TP 2 - Partie 3 et 4	17
	Exercice 1 : Exclusion mutuelle	17
1.1	Suspension d'une tâche	17
1.2	Réveil d'une tâche	18
1.3	Modèle de communications producteur/consommateur.	18
	Exercice 2 : Sémaphores	26
2.1	Implémentation	26
2.2	Producteur/Consommateur	30
2.3	2 Producteurs/ 1 Consommateur	33
2.4	1 Producteur/ 2 Consommateurs	40
2.5	2 Producteurs/ 2 Consommateurs	47
3	Rapport TP3 - Parties 5 et 6	56
	5ème partie : Le dîner des philosophes	56
	6ème partie : Communication par tubes	61
2.1	Fonction init_pipes	62
2.2	Fonction p_open	62

2.3	Fonction p_close	63
2.4	Fonction p_read	63
2.5	Fonction p_write	64
2.6	Test de ces fonctions	66

Table des figures

1.1	Résultat de testfile.c	9
2.1	Consommateur plus rapide que le producteur	22
2.2	Consommateur plus rapide que le producteur	23
2.3	Producteur plus rapide que le consommateur	24
2.4	Producteur plus rapide que le consommateur	25
2.5	2 producteurs plus rapides que le consommateur	38
2.6	Consommateur plus rapide que les 2 producteurs	39
2.7	Producteur plus rapide que les 2 consommateurs	45
2.8	2 consommateurs plus rapides que le producteur	46
2.9	2 producteurs et 2 consommateurs à la même vitesse	53
2.10	2 producteurs plus rapides que les 2 consommateurs	54
2.11	2 consommateurs plus rapides que les 2 producteurs	55
3.1	La table des philosophes	56
3.2	Le dîner des philosophes	60
3.3	Résultat du pipe	68

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.

```
1 void file_init ( void )
2 {
3     int i;
4     __queue=F_VIDE;
5
6     for ( i=0; i<MAX_TACHES; i++)
7         __file [ i]=F_VIDE;
8 }
```

Dans cette fonction, on se contente d'initialiser la queue de 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.

```
1 void ajoute ( uint16_t n )
2 {
3     if (n > MAX_TACHES - 1) // Si le numéro de la tâche est trop grand
4         return;
5
6     if (__queue == F_VIDE) // Si la file est vide
7     {
8         __queue = n;
9         __file [ __queue ] = n;
10    }
```

```
12  else // La file n'est pas vide
13  {
14      uint16_t tmp = _file[_queue];
15      _file[_queue] = n;
16      _file[n] = tmp;
17      _queue = n;
18  }
```

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 valoir 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.

```
2  uint16_t  suivant( void )
3  {
4      _queue = _file[_queue];
5      return _queue;
6  }
```

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 )
2 {
3     int i = 0;
4
5     if (t > MAX_TACHES || _queue == F_VIDE)
6         return;
7
8     while(_file[i] != t)
9         i++; // i = prédécesseur de t
10             // ex : ( 1 -> 0 -> 2 ) => avec t = 0, i = 1
11
12     if (i == MAX_TACHES)
13         return;
14
15     _file[i] = _file[t]; // 1 -> 2
16     _file[t] = F_VIDE;
17
18     if (t == _queue)
19     {
20         // si le prédécesseur est lui même => une seule tâche => queue =>
21         // on vide
22         if (t == i)
23             _queue = F_VIDE;
24         else
25             _queue = i; // la queue devient le prédécesseur de la tâche
26             // qu'on retire
27     }
28 }
```

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 prédécesseur 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.

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

    // Recherche du premier élément de la file
    while ( _file[i] == F_VIDE)
        i++;

    int temp = i; // temp ==> premier élément de la file

    // Tant qu'on a pas fait tout le tour de la file
    while ( _file[i] !=temp)
    {
        // Si l'élément i est la queue, afficher un "(Q)" à côté
        if (i==_queue)
            printf(" %d(Q) -> ", i);
        else
            printf(" %d -> ", i); // Sinon l'afficher normalement

        i=_file[i]; // Passer à l'élément suivant
    }
    // Affichage du dernier élément
    if (i==_queue)
        printf(" %d (Q) -> ", i);
    else
        printf(" %d -> ", i);

    printf("\n");
}
```

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 :

```
1 #include "noyau.h"
  #include "serialio.h"
3
5 int main()
  {
7     serial_init(115200);
9
10    printf("*****\n");
11
12    file_init();
13
14    // Ajouts des différentes tâches comme dans l'exemple du sujet
15    ajoute(3);
16    ajoute(5);
17    ajoute(1);
18    ajoute(0);
19    ajoute(2);
20
21    affic_file(); // Affichage de la file
22
23    affic_queue(); // Affichage de la queue
24    printf("Suivant() \n");
25    suivant(); // On fait un "suivant()"
26    affic_queue(); // On vérifie la queue après le "suivant()"
27
28    affic_file(); // Affichage de la file avec la nouvelle queue
29
30    printf("Retire 0 : \n");
31    retire(0); // On retire la tâche 0
32    affic_file(); // Affichage de la file après le "retire(0)"
33    printf("Ajoute 6 : \n");
34    ajoute(6); // On ajoute 6
35    affic_file(); // Affichage de la file après le "ajoute(6)"
36
37    return 0;
  }
```

A l'exécution, on obtient :

```
0 -> 2(Q) -> 3 -> 5 -> 1 ->
Queue : 2
Suivant()
Queue : 3
0 -> 2 -> 3(Q) -> 5 -> 1 ->
Retire 0 :
1 -> 2 -> 3(Q) -> 5 ->
Ajoute 6 :
1 -> 2 -> 3 -> 6(Q) -> 5 ->
```

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 :

```

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

```

2.1 Fonction noyau__exit()

```

1 void noyau__exit(void)
   {
3   int j;
     __irq_disable__(); // Désactiver les interruptions
5   printf("Sortie du noyau\n");
     for (j=0; j < MAX_TACHES; j++)
7     printf("\nActivations tache %d : %d", j, compteurs[j]);
     for (;;) // Terminer l'exécution
9   }

```

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()

```

1 void fin__tache(void)
   {
3   // on interdit les interruptions
     __irq_disable__();
5   // la tache est enlevee de la file des taches
     _contexte[_tache_c].status = CREE;
7   retire(_tache_c);
     schedule();

```

9 }

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 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)`

```

1 uint16_t cree( TACHE__ADR adr_tache )
2 {
3     CONTEXTE *p;                // pointeur d'une case de _contexte
4     static uint16_t tache = -1; // contient numero dernier cree
5
6
7     __lock__();                 // debut section critique
8     tache++;                    // numero de tache suivant
9
10    if (tache >= MAX_TACHES)     // sortie si depassement
11        noyau_exit();
12
13    p = &_contexte[tache];       // contexte de la nouvelle tache
14
15    p->sp_ini = _tos;             // allocation d'une pile a la tache
16    _tos -= PILE_TACHE + PILE_IRQ; // decrementation du pointeur de
17    pile pour la prochaine tache
18
19    __unlock__();                // fin section critique
20
21    p->tache_adr = adr_tache;     // memorisation adresse debut de
22    tache
23    p->status = CREE;            // mise a l'etat CREE
24    return(tache);              // tache est un uint16_t
25 }

```

La fonction *cree* permet de créer une tâche et de lui allouer une pile et un numéro. On commence tout d'abord par incrémenter le nombre de tâches créées via la variable *tache*. On récupère ensuite le contexte de la nouvelle tâche via le pointeur *p*. Le pointeur de pile associé à la tâche pointe à l'adresse du sommet de

pile (`_tos`) ce qui permet d'allouer une pile à la tâche. On décrémente ensuite le pointeur de pile par la taille maximale de la pile d'une tâche + la taille maximale de la pile IRQ par tâche. Ceci permet d'allouer assez de place pour la tâche suivante. Enfin, on enregistre l'adresse de début de la tâche et on passe le statut de la tâche à "CREE".

2.4 Fonction `active(uint16_t tache)`

```

1 void active( uint16_t tache )
2 {
3     CONTEXTE *p = &_contexte[tache]; // acces au contexte tache
4
5     if (p->status == NCREE)
6         noyau_exit(); // sortie du noyau
7
8     __lock__(); // debut section critique
9     if (p->status == CREE) // n'active que si receptif
10    {
11        p->status = PRET; // changement d'etat, mise a l'etat
12        PRET
13        ajoute(tache); // ajouter la tache dans la liste
14        schedule(); // activation d'une tache prete
15    }
16    __unlock__(); // fin section critique
17 }

```

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`

```

1 void __attribute__((naked)) scheduler( void )
2 {
3     register CONTEXTE *p;
4     register unsigned int sp asm("sp"); // Pointeur de pile
5
6     // Sauvegarder le contexte complet sur la pile IRQ
7     __asm__ __volatile__(
8         "stmfd sp, {r0-r14}^\t\n" // Sauvegarde registres mode system
9     );
10 }

```

```

10     "nop\t\n"           // Attendre un cycle
11     "sub    sp, sp, #60\t\n" // Ajustement pointeur de pile
12     "mrs    r0, spsr\t\n"   // Sauvegarde de spsr_irq
13     "stmfd  sp!, {r0, lr}\t\n"); // et de lr_irq
14
15     if (_ack_timer)        // Réinitialiser le timer si
16         nécessaire
17     {
18         register struct imx_timer* tim1 = (struct imx_timer *)
19         TIMER1_BASE;
20         tim1->tstat &=~TSTAT_COMP;
21     }
22     else
23     {
24         _ack_timer = 1;
25     }
26
27     _contexte[_tache_c].sp_irq = sp; // memoriser le pointeur de pile
28     _tache_c = suivant();           // recherche du suivant
29     if (_tache_c == F_VIDE)
30     {
31         printf("Plus rien à ordonnancer.\n");
32         noyau_exit();                // Sortie du noyau
33     }
34     compteurs[_tache_c]++;          // Incrémenter le compteur d'
35     activations
36     p = &_contexte[_tache_c];      // p pointe sur la nouvelle tache
37     courante
38
39     if (p->status == PRET)           // tache prete ?
40     {
41         sp = p->sp_ini;              // Charger sp_irq initial
42         _set_arm_mode_(ARMMODE_SYS); // Passer en mode système
43         sp = p->sp_ini - PILE_IRQ;    // Charger sp_sys initial
44         p->status = EXEC;             // status tache -> execution
45         _irq_enable_();              // autoriser les interruptions
46         (*p->tache_adr)();            // lancement de la tâche
47     }
48     else
49     {
50         sp = p->sp_irq;              // tache deja en execution ,
51         restaurer sp_irq
52     }
53
54     // Restaurer le contexte complet depuis la pile IRQ
55     __asm__ __volatile__(
56         "ldmfd  sp!, {r0, lr}\t\n" // Restaurer lr_irq
57         "msr    spsr, r0\t\n"      // et spsr_irq
58         "ldmfd  sp, {r0-r14}^\t\n" // Restaurer registres mode system

```

```

54     "nop\t\n"           // Attendre un cycle
    "add    sp, sp, #60\t\n" // Ajuster pointeur de pile irq
56     "subs  pc, lr, #4\t\n"); // Retour d'exception
}

```

Cette fonction fait office d'ordonnanceur de tâches. On rappelle que cette fonction s'exécute en mode IRQ. On commence tout d'abord par sauvegarder le contexte complet sur la pile IRQ car nous allons modifier le pointeur de pile *sp* du mode IRQ dans la fonction. On exécute ensuite la fonction *suivant()* pour récupérer la tâche à activer. Si cette tâche est vide alors il n'y a plus rien à ordonnancer donc on quitte le noyau. Sinon, on incrémente le compteur d'activations associé à cette tâche pour spécifier que cette tâche a été activée une fois de plus. Pour activer la tâche, il faut qu'elle soit en mode "PRET" sinon cela veut dire que la tâche est déjà en exécution. Dans ce deuxième cas, on met juste à jour le pointeur de pile *sp* du mode IRQ avec la valeur courante du pointeur de pile associé à la tâche. Dans le premier cas, si la tâche est en mode "PRET" alors il faut que le pointeur de pile *sp* en mode SYSTEM pointe sur la pile associée à la tâche. Pour cela, on passe en mode SYSTEM puis on affecte au pointeur de pile *sp* du mode SYSTEM la valeur initiale du pointeur de pile associée à la tâche - la taille maximale de la pile IRQ par tâche. On passe ensuite le statut de la tâche en "EXEC" pour qu'elle soit exécutable puis on exécute la tâche. Enfin, on restaure le contexte complet depuis la pile IRQ.

2.6 Fonction `schedule()`

```

void schedule( void )
2 {
    __lock__(); // Debut section critique
4
    // On simule une exception irq pour forcer un appel correct à
    scheduler().
6    __ack_timer = 0;
    __set_arm_mode__(ARMMODE_IRQ); // Passer en mode IRQ
8    __asm__ __volatile__(
        "mrs  r0, cpsr\t\n" // Sauvegarder cpsr dans spsr
10        "msr  spsr, r0\t\n"
        "add  lr, pc, #4\t\n" // Sauvegarder pc dans lr et
                                // l'ajuster
12        "b    scheduler\t\n" // Saut à scheduler
    );
14    __set_arm_mode__(ARMMODE_SYS); // Repasser en mode system
16    __unlock__(); // Fin section critique

```

 }

La fonction *schedule()* permet en fait de faire un appel à la fonction *scheduler()*. Dans une section critique, elle commence par passer en mode IRQ (car la fonction *scheduler* doit s'exécuter dans ce mode). Ensuite, en assembleur, on sauvegarde le registre **CPSR** dans **SPSR**. On rappelle que le registre **SPSR** est disponible dans chaque mode d'exception dont IRQ. Il est destiné à recevoir la sauvegarde du registre **CPSR** du programme interrompu. On poursuit en sauvegardant **pc** dans **lr** et on l'ajuste car après l'exécution de *scheduler* il faut revenir à ce niveau. On lance ensuite *scheduler* et on repasse en mode SYSTEM.

2.7 Fonction start(TACHE_ADR adr_tache)

```

1 void start( TACHE_ADR adr_tache )
2 {
3     short j;
4     register unsigned int sp asm("sp");
5     struct imx_timer* tim1 = (struct imx_timer *) TIMER1_BASE;
6     struct imx_aitc* aitc = (struct imx_aitc *) AITC_BASE;
7
8     for (j=0; j<MAX_TACHES; j++)
9     {
10         __contexte[j].status = NCREE;    // initialisation de l'etat des
11         taches
12     }
13     _tache_c = 0;                        // initialisation de la tache
14     courante
15     file_init();                         // initialisation de la file
16
17     _tos = sp;                           // Haut de la pile des tâches
18     _set_arm_mode__(ARMMODE_IRQ);        // Passer en mode IRQ
19     sp = _tos;                           // sp_irq initial
20     _set_arm_mode__(ARMMODE_SYS);        // Repasser en mode SYS
21
22     __irq_disable__();                   // on interdit les interruptions
23
24     // Initialisation du timer à 100 Hz
25     tim1->temp = 10000;
26     tim1->tprer = 0;
27     tim1->tctl |= TCTL_TEN | TCTL_IRQEN | TCTL_CLKSOURCE_PERCLK16;
28
29     // Initialisation de l'AITC
30     aitc->intenum = TIMER1_INT;

```



```
31  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. L'adresse du sommet de la pile (*_tos*) est ensuite mis à jour avec la valeur du pointeur de pile *sp* en mode SYSTEM car on s'apprête à lancer la première tâche. On passe ensuite en mode IRQ pour mettre à jour le pointeur de pile *sp* du mode IRQ qui lui pointe vers l'adresse du sommet de la pile. On repasse en mode SYSTEM puis on crée et on active la tâche pour lancer le système.

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 des 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()* :

```
1 void dort(void)
  {
3   __lock__();           // section critique

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

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

11  __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)
2 {
4     // on vérifie que la tâche existe et est suspendue
    if(t > MAX_TACHES || _contexte[t].status != SUSP)
6         return;
8     __lock__();    // section critique
10    CONTEXTE *p = &_contexte[t];
    p->status = EXEC;    // exécution
12
    ajoute(t);    // ajout dans la file
14    schedule();
16
    __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 (3 dans notre cas). 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 noter 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 }
65 }
67 fin_tache();
68 }
69 TACHE tacheConso(void)    // tâche consommateur
70 {
71     uint16_t k,i=0;
72
73     puts("————> EXEC tache Conso");
74
75     dort();    // consommateur dort initialement
76
77     while(1)
78     {
79         for (k=0; k<10000; k++);    // consommateur plus rapide
80
81         if (nb_places_libres==TAILLE_TABLEAU) // file vide
82         {
83             _lock_();
84             puts("**CONSO** -> Consommateur dort (Cas 3 : File vide)\n");
85             _unlock_();
86
87             dort();    // consommateur s'endort
88         }
89         else
90         {
91             _lock_();
92             printf("**CONSO** -> Lecture de fifo[%d] = %d\n", i, fifo[i]);
93             _unlock_();
94
95             nb_places_libres++;
96
97             i++;
98             if (i==TAILLE_TABLEAU)    // i = indice de la file
99                 i=0;
100         }
101
102         if (nb_places_libres==1) // file non pleine
103         {
104             _lock_();
105             puts("**CONSO** -> Reveil prod (Cas 4 : File non pleine)\n");
106             _unlock_();
107
108             reveille(prod);    // producteur se réveille
109         }
110     }
111 }
```

```
111     fin_tache();
113 }
115 int main()
116 {
117     serial_init(115200);
118     puts("Test noyau");
119     puts("Noyau preemptif");
120     puts("*****DEBUT*****\n\n\n\n\n");
121
122     printf("TAILLE DU TABLEAU : %d\n", TAILLE_TABLEAU);
123
124     start(tacheStart);
125     return(0);
126 }
```

Le résultat obtenu lorsque le consommateur est plus rapide que le producteur est donné sur la figure 2.2.

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 obtenons 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

```

TAILLE DU TABLEAU : 3
-----> EXEC tache Start
-----> EXEC tache Prod
-----> EXEC tache Conso
**PROD** -> Production : fifo[0] = 0
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**CONSO** -> Lecture de fifo[0] = 0
**CONSO** -> Consommateur dort (Cas 3 : File vide)

**PROD** -> Production : fifo[1] = 1
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**CONSO** -> Lecture de fifo[1] = 1
**CONSO** -> Consommateur dort (Cas 3 : File vide)

**PROD** -> Production : fifo[2] = 2
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**CONSO** -> Lecture de fifo[2] = 2
**PROD** -> Production : fifo[0] = 3
**CONSO** -> Consommateur dort (Cas 3 : File vide)

**PROD** -> Reveil du conso (Cas 2 : File non vide)

**CONSO** -> Lecture de fifo[0] = 3
**CONSO** -> Consommateur dort (Cas 3 : File vide)

**PROD** -> Production : fifo[1] = 4
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**CONSO** -> Lecture de fifo[1] = 4
**CONSO** -> Consommateur dort (Cas 3 : File vide)

**PROD** -> Production : fifo[2] = 5
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**CONSO** -> Lecture de fifo[2] = 5
**CONSO** -> Consommateur dort (Cas 3 : File vide)

**PROD** -> Production : fifo[0] = 6
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**CONSO** -> Lecture de fifo[0] = 6
**PROD** -> Production : fifo[1] = 7
**CONSO** -> Consommateur dort (Cas 3 : File vide)

**PROD** -> Reveil du conso (Cas 2 : File non vide)

**CONSO** -> Lecture de fifo[1] = 7
**CONSO** -> Consommateur dort (Cas 3 : File vide)

**PROD** -> Production : fifo[2] = 8
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**CONSO** -> Lecture de fifo[2] = 8
**CONSO** -> Consommateur dort (Cas 3 : File vide)

**PROD** -> Production : fifo[0] = 9
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**CONSO** -> Lecture de fifo[0] = 9
**CONSO** -> Consommateur dort (Cas 3 : File vide)

**PROD** -> Production : fifo[1] = 10
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**CONSO** -> Lecture de fifo[1] = 10
**PROD** -> Production : fifo[2] = 11
**CONSO** -> Consommateur dort (Cas 3 : File vide)

```

FIGURE 2.2 – 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 2.3 – Producteur plus rapide que le consommateur

```

TAILLE DU TABLEAU : 3
-----> EXEC tache Start
-----> EXEC tache Prod
-----> EXEC tache Conso
**PROD** -> Production : fifo[0] = 0
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**PROD** -> Production : fifo[1] = 1
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**CONSO** -> Lecture de fifo[0] = 0
**PROD** -> Production : fifo[2] = 2
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**PROD** -> Production : fifo[0] = 3
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**CONSO** -> Lecture de fifo[1] = 1
**CONSO** -> Reveil prod (Cas 4 : File non pleine)

**PROD** -> Production : fifo[1] = 4
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**PROD** -> Producteur dort (Cas 1 : File pleine)
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**PROD** -> Producteur dort (Cas 1 : File pleine)

**CONSO** -> Lecture de fifo[2] = 2
**CONSO** -> Reveil prod (Cas 4 : File non pleine)

**PROD** -> Reveil du conso (Cas 2 : File non vide)

**PROD** -> Production : fifo[2] = 5
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**PROD** -> Producteur dort (Cas 1 : File pleine)

**CONSO** -> Lecture de fifo[0] = 3
**CONSO** -> Reveil prod (Cas 4 : File non pleine)

**PROD** -> Reveil du conso (Cas 2 : File non vide)

**PROD** -> Production : fifo[0] = 6
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**PROD** -> Producteur dort (Cas 1 : File pleine)

**CONSO** -> Lecture de fifo[1] = 4
**CONSO** -> Reveil prod (Cas 4 : File non pleine)

**PROD** -> Reveil du conso (Cas 2 : File non vide)

**PROD** -> Production : fifo[1] = 7
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**PROD** -> Producteur dort (Cas 1 : File pleine)

**CONSO** -> Lecture de fifo[2] = 5
**CONSO** -> Reveil prod (Cas 4 : File non pleine)

**PROD** -> Reveil du conso (Cas 2 : File non vide)

**PROD** -> Production : fifo[2] = 8
**PROD** -> Reveil du conso (Cas 2 : File non vide)

**PROD** -> Producteur dort (Cas 1 : File pleine)

**CONSO** -> Lecture de fifo[0] = 6
**CONSO** -> Reveil prod (Cas 4 : File non pleine)

```

FIGURE 2.4 – Producteur plus rapide que le consommateur

La méthode d'exclusion mutuelle présentée précédemment avec endormissement et réveil des tâches peut poser problème dans le cas général. En effet, nous ne sommes pas à l'abri d'un interblocage des deux tâches si elles sont toutes les deux endormies et qu'elles attendent d'être réveillées. Ceci peut arriver puisque rien ne garantit qu'un signal de réveil ne soit pas interrompue. Une première tâche pourrait en principe envoyer son signal de réveil avant que la seconde tâche s'endorme ce qui engendrerait une attente infinie.

Exercice 2 : Sémaphores

Un deuxième outil plus efficace que le précédent pour gérer les problèmes de ressources partagées repose sur les sémaphores. Ce mécanisme permet de gérer l'envoi des signaux indépendamment de l'exécution des tâches.

2.1 Implémentation

Un sémaphore est composé d'une FIFO de tâches en attente et d'une valeur représentant le nombre de ressources disponibles. Une FIFO est une structure circulaire composée d'une file de tâches, de l'indice de début de la file et de l'indice de fin de la file. Dans cet exercice, nous considérons un tableau de sémaphores. Ci-dessous le fichier header *sem.h* :

```
1 #ifndef SEM_H_
2 #define SEM_H_
4 #include <stdint.h>
6 #define MAX_SEM 15
8 typedef struct {
9     short  taches[MAX_TACHES];
10    short  debut_file;
11    short  fin_file;
12 } FIFO;
14 typedef struct {
15     FIFO file; // File circulaire des tâches en attente
16     short valeur; // compteur du sémaphore e(s)
17 } SEMAPHORE;
18
19 void s__init( void );
20 ushort s__cree( short v );
21 void s__close( ushort n );
```

```
22 void s_wait( ushort n );  
void s_signal ( ushort n );  
24 #endif /* SEM_H */
```

Un sémaphore se manipule via les cinq fonctions suivantes : *s_init*, *s_cree*, *s_close*, *s_wait* et *s_signal*, toutes définies dans le fichier **sem.c**.

Tout d'abord la fonction *s_init* permet d'initialiser la valeur de notre tableau de sémaphores. Ici, nous avons choisi d'initialiser toutes nos sémaphores à -1000, nombre que nous considérons, dans cet exercice, inatteignable.

```
1 SEMAPHORE _sem[MAX_SEM];  
  
3 void s_init( void )  
{  
5     int i;  
  
7     for (i = 0; i < MAX_SEM; i++)  
        {  
9         _sem[i].valeur = -1000;    // non crée  
        }  
11 }
```

La seconde fonction *s_cree* nous sert à créer un sémaphore et à l'initialiser à une certaine valeur donnée.

```
1 ushort s_cree( short v )  
{  
3     int i = 0;  
  
5     // si la valeur du sémaphore dépasse le nombre de tâches  
    if (v >= MAX_TACHES)  
7         return -1;  
  
9     // on cherche un emplacement libre  
    while (i < MAX_SEM && _sem[i].valeur != -1000)  
11         i++;  
  
13     if (i >= MAX_SEM)    // si on n'a pas trouver d'emplacement libre  
        return -1;  
  
15     // initialisation du sémaphore à la valeur v
```

```
17  _sem[i].valeur = v;  
    _sem[i].file.debut_file = 0;  
19  _sem[i].file.fin_file = 0;  
  
21  return i;  
}
```

La troisième fonction *s_close* réinitialise un sémaphore donné. La valeur du sémaphore est remise à -1000 pour qu'on le considère non créé.

```
void s_close( ushort n )  
{  
    if(n >= MAX_SEM || n < 0)  
        return;  
  
    _sem[n].valeur = -1000;  
}
```

La fonction *s_wait* implémente l'opération P qui permet de prendre une ressource associée au sémaphore. On décrémente donc la valeur du sémaphore et si celle-ci est strictement inférieure à 0 alors on bloque/endort la tâche ayant effectué la requête et on la met dans la file des tâches en attente associée au sémaphore. Dans notre fonction les tâches sont ajoutées en fin de file.

```
1  void s_wait( ushort n )  
    {  
3      if(n >= MAX_SEM || n < 0)  
          return;  
5  
        // si le sémaphore n'est pas créé  
7        if(_sem[n].valeur == -1000)  
            return;  
9  
        // zone critique car plusieurs tâches ne doivent pas modifier le  
        // sémaphore en même temps  
11       _lock_();  
  
13       _sem[n].valeur--;          // décrémentation du sémaphore  
  
15       if(_sem[n].valeur < 0)  
          {  
17          // on ajoute la tâche courante à la fin de la file  
            _sem[n].file.taches[_sem[n].file.fin_file] = _tache_c;
```

```
19 // incrémente la fin de la file
21 _sem[n].file.fin_file = _sem[n].file.fin_file + 1;

23 // on met à jour la fin de la file circulaire
// si fin_file < MAX_TACHES alors on ne change rien sinon on
// revient au début de la file
25 _sem[n].file.fin_file = (_sem[n].file.fin_file) % MAX_TACHES;

27 dort(); // la tâche courante s'endort
}

29 _unlock();
31 }
```

Enfin, la fonction *s_signal* implémente l'opération V qui permet de libérer une ressource associée au sémaphore. On incrémente donc la valeur du sémaphore et si celle-ci est inférieure ou égale à 0 alors on extrait une tâche de la file des tâches en attente et on la libère/réveille. Dans notre fonction les tâches sont extraites au début de file.

```
1 void s_signal ( ushort n )
{
3     short tache;

5     if(n >= MAX_SEM || n < 0)
        return;

7     // si le sémaphore n'est pas créé
9     if(_sem[n].valeur == -1000)
        return;

11    // zone critique car plusieurs tâches ne doivent pas modifier le
// sémaphore en même temps
13    _lock();

15    _sem[n].valeur++; // incrémentation du sémaphore

17    if(_sem[n].valeur <= 0)
    {
19        // on extrait une tâche au début de la file
        tache = _sem[n].file.taches[_sem[n].file.debut_file];

21        // incrémente le début de la file
23        _sem[n].file.debut_file = _sem[n].file.debut_file + 1;
```

```

25 // on met à jour le début de la file circulaire
26 // si debut_file < MAX_TACHES alors on ne change rien sinon on va
27 // à la fin de la file
28 _sem[n].file.debut_file = (_sem[n].file.debut_file) % MAX_TACHES;
29
30 reveille(tache); // réveil de la tâche
31 }
32
33 _unlock();
34 }

```

2.2 Producteur/Consommateur

Afin de tester nos sémaphores, nous avons ré-implémenté le problème du producteur/consommateur en remplaçant les primitives *dort()* et *veille()* par respectivement *s_wait* et *s_signal*. Nous mettons en place deux sémaphores associés respectivement au producteur et au consommateur qu'on initialise à 0. Ceci permet au producteur d'être bloqué si la file est pleine et au consommateur d'être réveillé si il y a un entier à consommer. Le principe reste exactement le même que dans l'exercice précédent.

```

1 #define TAILLE_TABLEAU 3
2
3 TACHE tacheStart();
4 TACHE tacheProd();
5 TACHE tacheConso();
6 uint16_t prod, conso;
7 uint16_t fifo[TAILLE_TABLEAU]; // la file
8 uint16_t nb_places_libres = TAILLE_TABLEAU;
9 ushort sem1, sem2; // sem1 => producteur, sem2 => consommateur
10
11 TACHE tacheStart(void) // tâche de démarrage
12 {
13     puts("————> EXEC tache Start");
14     prod = cree(tacheProd);
15     conso = cree(tacheConso);
16
17     active(prod);
18     active(conso);
19
20     fin_tache();
21 }
22
23 TACHE tacheProd(void) // tâche producteur
24 {

```

```

25 puts("————> EXEC tache Prod");
27 uint16_t j=0,k,i=0;
29 while(1)
30 {
31     for (k=0; k<30000; k++);    // producteur plus lent
33     if (nb_places_libres>=1)    // production d'un entier
34     {
35         _lock_();                // zone critique
36         printf("**PROD** -> Production : fifo[%d] = %d\n", i, j);
37         fifo[i]=j;
38         _unlock_();
39         j++;
41         nb_places_libres--;
43         i++;
44         if (i==TAILLE_TABLEAU) // i = indice de la file
45             i=0;
46     }
47     else // file pleine
48     {
49         _lock_();
50         puts("**PROD** -> Producteur dort (Cas 1 : File pleine)\n");
51         _unlock_();
53         s_wait(sem1);    // dort()
54     }
55     if (nb_places_libres<TAILLE_TABLEAU) // file non vide
56     {
57         _lock_();
58         puts("**PROD** -> Reveil du conso (Cas 2 : File non vide)\n");
59         _unlock_();
61         s_signal(sem2);    // reveille(conso);
62     }
63 }
65 }
67 fin_tache();
68 }
69 TACHE tacheConso(void)    // tâche consommateur
70 {
71     uint16_t k,i=0;
72 }
73

```



```

puts("————> EXEC tache Conso");
75
s_wait(sem2);    // dort()
77
while(1)
79 {
    for (k=0; k<10000; k++); // consommateur plus rapide
81
    if (nb_places_libres==TAILLE_TABLEAU) // file vide
83 {
        _lock_();
85 puts("**CONSO** -> Consommateur dort (Cas 3 : File vide)\n");
        _unlock_();

87 s_wait(sem2);    // dort()
89 }
    else
91 {
        _lock_();
93 printf("**CONSO** -> Lecture de fifo[%d] = %d\n", i, fifo[i]);
        _unlock_();

95 nb_places_libres++;

97 i++;
99 if (i==TAILLE_TABLEAU) // i = indice de la file
        i=0;
101 }

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

109 s_signal(sem1);    // reveille(prod);
    }
111 }

113 fin_tache();
}
115
117 int main()
{
    serial_init(115200);
119 puts("Test noyau");
    puts("Noyau preemptif");
121 puts("*****DEBUT*****\n\n\n\n\n");

```

```

123 printf("TAILLE DU TABLEAU : %d\n", TAILLE_TABLEAU);
125
125 s_init();
125 sem1 = s_cree(0); // prod
127 sem2 = s_cree(0); // conso
127 start(tacheStart);
129
129 return(0);
131 }

```

Le résultat obtenu lorsque le consommateur est plus rapide que le producteur est exactement le même que sur la figure 2.2. De même, dans le cas où le producteur est plus rapide que le consommateur, nous obtenons exactement le même résultat que dans la figure 2.4.

2.3 2 Producteurs/ 1 Consommateur

Nous avons également testé nos sémaphores en ajoutant un producteur au problème précédent. Pour cela, il faut mettre en place un sémaphore binaire ou mutex qu'on initialise à 1 et qui empêche les deux producteurs de produire en même temps. On bloque le mutex avant production et on le libère après. De plus, il faut mettre en place des variables globales pour les indices et les valeurs de la file pour que les producteurs manipulent les mêmes données.

```

1 #define TAILLE_TABLEAU 3
3
3 TACHE tacheStart();
3 TACHE tacheProd1();
5 TACHE tacheProd2();
5 TACHE tacheConso();
7
7 uint16_t prod1, prod2, conso; // 2 producteurs et 1 consommateur
7 uint16_t fifo[TAILLE_TABLEAU];
9
9 uint16_t nb_places_libres = TAILLE_TABLEAU;
9 ushort sem1, sem2, semMutexProd;
11
11 // variables globales pour les producteurs
13 // l => indice de la file et j => valeur à cet indice
13 int l = 0, j = 0;
15
15 TACHE tacheStart(void)
17 {
17 puts("————> EXEC tache Start");
19

```

```

21 prod1 = cree(tacheProd1);
22 prod2 = cree(tacheProd2);
23 conso = cree(tacheConso);
24
25 active(conso);
26 active(prod1);
27 active(prod2);
28
29 fin_tache();
30 }
31
32 TACHE tacheProd1(void)
33 {
34     puts("————> EXEC tache Prod 1");
35
36     uint16_t k;
37
38     while(1)
39     {
40         //for (k=0; k<30000; k++);
41         for (k=0; k<10000; k++);
42
43         s_wait(semMutexProd); // P du mutex (entrée de la zone critique)
44
45         if (nb_places_libres>=1)
46         {
47             _lock_();
48             printf("**PROD 1** -> Production : fifo[%d] = %d\n", l, j);
49             fifo[l]=j;
50             _unlock_();
51             j++;
52
53             nb_places_libres--;
54
55             l++;
56             if (l==TAILLE_TABLEAU)
57                 l=0;
58         }
59         else
60         {
61             s_wait(sem1); // dort()
62         }
63
64         s_signal(semMutexProd); // V du mutex (sortie de la zone critique)
65
66         if (nb_places_libres<TAILLE_TABLEAU)
67         {
68             s_signal(sem2); // reveille(conso);

```

```

69     }
71 }
73 fin_tache();
75 }
77 TACHE tacheProd2(void)
78 {
79     puts("————> EXEC tache Prod 2");
81     uint16_t k;
83     while(1)
84     {
85         //for (k=0; k<30000; k++);
86         for (k=0; k<10000; k++);
88         s_wait(semMutexProd); // P du mutex (entrée de la zone critique)
90         if (nb_places_libres>=1)
91         {
92             _lock_();
93             printf("**PROD 2** -> Production : fifo[%d] = %d\n", l, j);
94             fifo[l]=j;
95             _unlock_();
96             j++;
98             nb_places_libres--;
100             l++;
101             if (l==TAILLE_TABLEAU)
102                 l=0;
103         }
104         else
105         {
106             s_wait(sem1); // dort()
107         }
109         s_signal(semMutexProd); // V du mutex (sortie de la zone critique)
111         if (nb_places_libres<TAILLE_TABLEAU)
112         {
113             s_signal(sem2); // reveille(conso);
114         }
115     }
117     fin_tache();

```

```

}
119
TACHE tacheConso(void)
121 {
    uint16_t k,i=0;
123
    puts("————> EXEC tache Conso");
125
    s_wait(sem2); // dort()
127
    while(1)
129 {
        for (k=0; k<30000; k++);
131 //for (k=0; k<10000; k++);

        if (nb_places_libres==TAILLE_TABLEAU)
133 {
            s_wait(sem2); // dort()
135 }
        else
137 {
            _lock_();
139 printf("**CONSO** -> Lecture de fifo[%d] = %d\n", i, fifo[i]);
141 _unlock_();

            nb_places_libres++;
143
            i++;
            if (i==TAILLE_TABLEAU)
145 i=0;
147 }

        if (nb_places_libres==1)
149 {
            s_signal(sem1); // reveille(prod);
151 }
153 }

155 }

157 fin_tache();
159 }

int main()
161 {
    serial_init(115200);
163 puts("Test noyau");
    puts("Noyau preemptif");
165 puts("*****DEBUT*****\n\n\n\n\n");

```

```
167 printf("TAILLE DU TABLEAU : %d\n", TAILLE_TABLEAU);
169 s_init();
171 sem1 = s_cree(0); // prod
171 sem2 = s_cree(0); // conso
173 // mutex entre les 2 producteurs pour les empêcher de produire en
    même temps
175 semMutexProd = s_cree(1);
177 start(tacheStart);
179 return(0);
}
```

Nous avons testé deux configurations, l'une où les producteurs sont plus rapides que le consommateur et la deuxième où le consommateur est plus rapide que les producteurs. On suppose que les producteurs ont la même vitesse.

Le résultat dans le premier cas est visible sur la figure 2.5. On voit bien dans ce cas que le producteur 1 et 2 ne produisent pas en même temps mais à tour de rôle. Comme le consommateur est plus lent il consomme les entiers avec un retard de 2 indices. Par exemple, le producteur 2 écrit l'entier 2 à la case 2 et juste après le consommateur lit l'entier 0 à la case 0 (2 écritures ont été faite entre temps à cause de la rapidité des producteurs).

```

TAILLE DU TABLEAU : 3
-----> EXEC tache Start
-----> EXEC tache Conso
-----> EXEC tache Prod 1
-----> EXEC tache Prod 2
**PROD 2** -> Production : fifo[0] = 0
**PROD 1** -> Production : fifo[1] = 1
**PROD 2** -> Production : fifo[2] = 2
**CONSO** -> Lecture de fifo[0] = 0
**PROD 2** -> Production : fifo[0] = 3
**CONSO** -> Lecture de fifo[1] = 1
**PROD 2** -> Production : fifo[1] = 4
**CONSO** -> Lecture de fifo[2] = 2
**PROD 2** -> Production : fifo[2] = 5
**CONSO** -> Lecture de fifo[0] = 3
**PROD 2** -> Production : fifo[0] = 6
**CONSO** -> Lecture de fifo[1] = 4
**PROD 2** -> Production : fifo[1] = 7
**CONSO** -> Lecture de fifo[2] = 5
**PROD 2** -> Production : fifo[2] = 8
**CONSO** -> Lecture de fifo[0] = 6
**PROD 2** -> Production : fifo[0] = 9
**CONSO** -> Lecture de fifo[1] = 7
**CONSO** -> Lecture de fifo[2] = 8
**PROD 2** -> Production : fifo[1] = 10
**PROD 1** -> Production : fifo[2] = 11
**CONSO** -> Lecture de fifo[0] = 9
**PROD 1** -> Production : fifo[0] = 12
**CONSO** -> Lecture de fifo[1] = 10
**PROD 1** -> Production : fifo[1] = 13
**CONSO** -> Lecture de fifo[2] = 11
**PROD 1** -> Production : fifo[2] = 14
**CONSO** -> Lecture de fifo[0] = 12
**PROD 1** -> Production : fifo[0] = 15
**CONSO** -> Lecture de fifo[1] = 13
**PROD 1** -> Production : fifo[1] = 16
**CONSO** -> Lecture de fifo[2] = 14
**PROD 1** -> Production : fifo[2] = 17
**CONSO** -> Lecture de fifo[0] = 15
**PROD 1** -> Production : fifo[0] = 18
**CONSO** -> Lecture de fifo[1] = 16
**CONSO** -> Lecture de fifo[2] = 17
**PROD 1** -> Production : fifo[1] = 19
**PROD 2** -> Production : fifo[2] = 20
**CONSO** -> Lecture de fifo[0] = 18
**PROD 2** -> Production : fifo[0] = 21
**CONSO** -> Lecture de fifo[1] = 19
**PROD 2** -> Production : fifo[1] = 22
**CONSO** -> Lecture de fifo[2] = 20
**PROD 2** -> Production : fifo[2] = 23
**CONSO** -> Lecture de fifo[0] = 21
**PROD 2** -> Production : fifo[0] = 24
**CONSO** -> Lecture de fifo[1] = 22
**PROD 2** -> Production : fifo[1] = 25
**CONSO** -> Lecture de fifo[2] = 23
**PROD 2** -> Production : fifo[2] = 26
**CONSO** -> Lecture de fifo[0] = 24
**PROD 2** -> Production : fifo[0] = 27
**CONSO** -> Lecture de fifo[1] = 25
**CONSO** -> Lecture de fifo[2] = 26

```

FIGURE 2.5 – 2 producteurs plus rapides que le consommateur

Dans le cas où le consommateur est plus rapide (figure 2.6), on remarque qu'il n'a pas de retard, il consomme les entiers directement après production ce qui est tout à fait cohérent.

```

TAILLE DU TABLEAU : 3
-----> EXEC tache Start
-----> EXEC tache Prod 1
-----> EXEC tache Prod 2
-----> EXEC tache Conso
**PROD 2** -> Production : fifo[0] = 0
**CONSO** -> Lecture de fifo[0] = 0
**PROD 1** -> Production : fifo[1] = 1
**PROD 2** -> Production : fifo[2] = 2
**CONSO** -> Lecture de fifo[1] = 1
**CONSO** -> Lecture de fifo[2] = 2
**PROD 1** -> Production : fifo[0] = 3
**CONSO** -> Lecture de fifo[0] = 3
**PROD 2** -> Production : fifo[1] = 4
**CONSO** -> Lecture de fifo[1] = 4
**PROD 1** -> Production : fifo[2] = 5
**CONSO** -> Lecture de fifo[2] = 5
**PROD 2** -> Production : fifo[0] = 6
**CONSO** -> Lecture de fifo[0] = 6
**PROD 1** -> Production : fifo[1] = 7
**CONSO** -> Lecture de fifo[1] = 7
**PROD 2** -> Production : fifo[2] = 8
**CONSO** -> Lecture de fifo[2] = 8
**PROD 1** -> Production : fifo[0] = 9
**CONSO** -> Lecture de fifo[0] = 9
**PROD 2** -> Production : fifo[1] = 10
**CONSO** -> Lecture de fifo[1] = 10
**PROD 1** -> Production : fifo[2] = 11
**PROD 2** -> Production : fifo[0] = 12
**CONSO** -> Lecture de fifo[2] = 11
**CONSO** -> Lecture de fifo[0] = 12
**PROD 1** -> Production : fifo[1] = 13
**PROD 2** -> Production : fifo[2] = 14
**CONSO** -> Lecture de fifo[1] = 13
**CONSO** -> Lecture de fifo[2] = 14
**PROD 1** -> Production : fifo[0] = 15
**CONSO** -> Lecture de fifo[0] = 15
**PROD 2** -> Production : fifo[1] = 16
**CONSO** -> Lecture de fifo[1] = 16
**PROD 1** -> Production : fifo[2] = 17
**CONSO** -> Lecture de fifo[2] = 17
**PROD 2** -> Production : fifo[0] = 18
**CONSO** -> Lecture de fifo[0] = 18
**PROD 1** -> Production : fifo[1] = 19
**CONSO** -> Lecture de fifo[1] = 19
**PROD 2** -> Production : fifo[2] = 20
**CONSO** -> Lecture de fifo[2] = 20
**PROD 1** -> Production : fifo[0] = 21
**CONSO** -> Lecture de fifo[0] = 21
**PROD 2** -> Production : fifo[1] = 22
**CONSO** -> Lecture de fifo[1] = 22
**PROD 1** -> Production : fifo[2] = 23
**PROD 2** -> Production : fifo[0] = 24
**CONSO** -> Lecture de fifo[2] = 23
**CONSO** -> Lecture de fifo[0] = 24
**PROD 1** -> Production : fifo[1] = 25
**PROD 2** -> Production : fifo[2] = 26
**CONSO** -> Lecture de fifo[1] = 25
**PROD 1** -> Production : fifo[0] = 27
**CONSO** -> Lecture de fifo[2] = 26
**CONSO** -> Lecture de fifo[0] = 27
**PROD 2** -> Production : fifo[1] = 28
**CONSO** -> Lecture de fifo[1] = 28

```

FIGURE 2.6 – Consommateur plus rapide que les 2 producteurs

2.4 1 Producteur/ 2 Consommateurs

Nous avons également testé le cas opposé où il y a cette fois-ci un producteur et deux consommateurs. Le principe est exactement le même que précédemment sauf que le mutex s'applique aux deux consommateurs avant et après lecture d'un entier. De même que précédemment, les consommateurs partagent des variables globales. On suppose que les consommateurs ont la même vitesse.

```

1 #define TAILLE_TABLEAU 3
3 TACHE tacheStart();
3 TACHE tacheProd();
5 TACHE tacheConso1();
5 TACHE tacheConso2();
7 uint16_t prod, conso1, conso2;    // 2 conso et 1 prod
7 uint16_t fifo[TAILLE_TABLEAU];
9 uint16_t nb_places_libres = TAILLE_TABLEAU;
9 ushort sem1, sem2, semMutexConso;
11
13 // variables globales pour les producteurs
13 // l => indice de la file et j => valeur à cet indice
13 int l = 0, j = 0;
15
15 TACHE tacheStart(void)
17 {
17     puts("————> EXEC tache Start");
19
19     prod = cree(tacheProd);
21     conso1 = cree(tacheConso1);
21     conso2 = cree(tacheConso2);
23
23     active(conso1);
25     active(conso2);
25     active(prod);
27
27     fin_tache();
29 }
31 TACHE tacheProd(void)
31 {
33     puts("————> EXEC tache Prod");
35
35     uint16_t j=0,k,i=0;
37
37     while(1)
37     {
39         for (k=0; k<30000; k++);

```

```

41 //for (k=0; k<10000; k++);
42
43 if (nb_places_libres >= 1)
44 {
45     _lock_();
46     printf(" **PROD** -> Production : fifo[%d] = %d\n", i, j);
47     fifo[i] = j;
48     _unlock_();
49     j++;
50
51     nb_places_libres--;
52
53     i++;
54     if (i == TAILLE_TABLEAU)
55         i = 0;
56 }
57 else
58 {
59     s_wait(sem1); // dort()
60 }
61
62 if (nb_places_libres < TAILLE_TABLEAU)
63 {
64     s_signal(sem2); // reveille(conso);
65 }
66
67 }
68
69 fin_tache();
70
71 TACHE tacheConso1(void)
72 {
73     uint16_t k;
74
75     puts("—————> EXEC tache Conso 1");
76
77     s_wait(sem2); // dort()
78
79     while(1)
80     {
81         //for (k=0; k<30000; k++);
82         for (k=0; k<10000; k++);
83
84
85         s_wait(semMutexConso); // P du mutex (entrée de la zone critique)
86
87         if (nb_places_libres == TAILLE_TABLEAU)

```

```

89     {
90         s__wait(sem2); // dort()
91     }
92     else
93     {
94         __lock__();
95         printf("**CONSO 1** -> Lecture de fifo[%d] = %d\n", l, fifo[l]);
96         __unlock__();
97
98         nb_places_libres++;
99
100        l++;
101        if (l==TAILLE_TABLEAU)
102            l=0;
103    }
104
105    s__signal(semMutexConso); // V du mutex (sortie de la zone critique)
106
107    if (nb_places_libres==1)
108    {
109        s__signal(sem1); // reveille(prod);
110    }
111
112    }
113
114    fin_tache();
115 }
116
117 TACHE tacheConso2(void)
118 {
119     uint16_t k;
120
121     puts("—————> EXEC tache Conso 2");
122
123     s__wait(sem2); // dort()
124
125     while(1)
126     {
127         //for (k=0; k<30000; k++);
128         for (k=0; k<10000; k++);
129
130
131         s__wait(semMutexConso); // P du mutex (entrée de la zone critique)
132
133         if (nb_places_libres==TAILLE_TABLEAU)
134         {
135             s__wait(sem2); // dort()
136         }

```

```
137     else
138     {
139         _lock_();
140         printf("**CONSO 2** -> Lecture de fifo[%d] = %d\n", l, fifo[l]);
141         _unlock_();
142
143         nb_places_libres++;
144
145         l++;
146         if (l==TAILLE_TABLEAU)
147             l=0;
148     }
149
150     s_signal(semMutexConso); // V du mutex (sortie de la zone critique)
151
152
153     if (nb_places_libres==1)
154     {
155         s_signal(sem1); // reveille(prod);
156     }
157 }
158
159 fin_tache();
160 }
161
162 int main()
163 {
164     serial_init(115200);
165     puts("Test noyau");
166     puts("Noyau preemptif");
167     puts("*****DEBUT*****\n\n\n\n\n");
168
169     printf("TAILLE DU TABLEAU : %d\n", TAILLE_TABLEAU);
170
171     s_init();
172     sem1 = s_cree(0); // prod
173     sem2 = s_cree(0); // conso
174
175     // mutex entre les 2 consommateurs pour les empêcher de consommer
176     // en même temps
177     semMutexConso = s_cree(1);
178     start(tacheStart);
179     return(0);
180 }
```

Le résultat dans le cas où le producteur est plus rapide que les consommateurs est visible sur la figure 2.7. Les consommateurs ont tendance à être en retard d'un ou deux indices.

Pour le cas où les consommateurs sont plus rapides que le producteur (figure 2.8), la consommation se fait dans l'ordre juste après les productions comme précédemment.

```

TAILLE DU TABLEAU : 3
-----> EXEC tache Start
-----> EXEC tache Conso 1
-----> EXEC tache Conso 2
-----> EXEC tache Prod
**PROD** -> Production : fifo[0] = 0
**PROD** -> Production : fifo[1] = 1
**CONSO 1** -> Lecture de fifo[0] = 0
**PROD** -> Production : fifo[2] = 2
**PROD** -> Production : fifo[0] = 3
**CONSO 2** -> Lecture de fifo[1] = 1
**CONSO 1** -> Lecture de fifo[2] = 2
**PROD** -> Production : fifo[1] = 4
**PROD** -> Production : fifo[2] = 5
**CONSO 2** -> Lecture de fifo[0] = 3
**CONSO 1** -> Lecture de fifo[1] = 4
**PROD** -> Production : fifo[0] = 6
**PROD** -> Production : fifo[1] = 7
**CONSO 2** -> Lecture de fifo[2] = 5
**CONSO 1** -> Lecture de fifo[0] = 6
**PROD** -> Production : fifo[2] = 8
**PROD** -> Production : fifo[0] = 9
**CONSO 2** -> Lecture de fifo[1] = 7
**CONSO 1** -> Lecture de fifo[2] = 8
**PROD** -> Production : fifo[1] = 10
**PROD** -> Production : fifo[2] = 11
**CONSO 2** -> Lecture de fifo[0] = 9
**PROD** -> Production : fifo[0] = 12
**CONSO 1** -> Lecture de fifo[1] = 10
**CONSO 2** -> Lecture de fifo[2] = 11
**PROD** -> Production : fifo[1] = 13
**PROD** -> Production : fifo[2] = 14
**CONSO 1** -> Lecture de fifo[0] = 12
**PROD** -> Production : fifo[0] = 15
**CONSO 2** -> Lecture de fifo[1] = 13
**PROD** -> Production : fifo[1] = 16
**CONSO 1** -> Lecture de fifo[2] = 14
**PROD** -> Production : fifo[2] = 17
**CONSO 2** -> Lecture de fifo[0] = 15
**PROD** -> Production : fifo[0] = 18
**CONSO 1** -> Lecture de fifo[1] = 16
**PROD** -> Production : fifo[1] = 19
**CONSO 2** -> Lecture de fifo[2] = 17
**PROD** -> Production : fifo[2] = 20
**CONSO 1** -> Lecture de fifo[0] = 18
**PROD** -> Production : fifo[0] = 21
**CONSO 2** -> Lecture de fifo[1] = 19
**PROD** -> Production : fifo[1] = 22
**CONSO 1** -> Lecture de fifo[2] = 20
**CONSO 2** -> Lecture de fifo[0] = 21
**PROD** -> Production : fifo[2] = 23
**PROD** -> Production : fifo[0] = 24
**CONSO 1** -> Lecture de fifo[1] = 22
**CONSO 2** -> Lecture de fifo[2] = 23
**PROD** -> Production : fifo[1] = 25
**PROD** -> Production : fifo[2] = 26
**CONSO 1** -> Lecture de fifo[0] = 24
**PROD** -> Production : fifo[0] = 27
**CONSO 2** -> Lecture de fifo[1] = 25
**PROD** -> Production : fifo[1] = 28
**CONSO 1** -> Lecture de fifo[2] = 26
**PROD** -> Production : fifo[2] = 29
**CONSO 2** -> Lecture de fifo[0] = 27
**PROD** -> Production : fifo[0] = 30

```

FIGURE 2.7 – Producteur plus rapide que les 2 consommateurs

```

TAILLE DU TABLEAU : 3
-----> EXEC tache Start
-----> EXEC tache Conso 1
-----> EXEC tache Conso 2
-----> EXEC tache Prod
**PROD** -> Production : fifo[0] = 0
**CONSO 1** -> Lecture de fifo[0] = 0
**PROD** -> Production : fifo[1] = 1
**PROD** -> Production : fifo[2] = 2
**PROD** -> Production : fifo[0] = 3
**CONSO 2** -> Lecture de fifo[1] = 1
**CONSO 1** -> Lecture de fifo[2] = 2
**CONSO 2** -> Lecture de fifo[0] = 3
**PROD** -> Production : fifo[1] = 4
**CONSO 1** -> Lecture de fifo[1] = 4
**PROD** -> Production : fifo[2] = 5
**CONSO 1** -> Lecture de fifo[2] = 5
**PROD** -> Production : fifo[0] = 6
**PROD** -> Production : fifo[1] = 7
**CONSO 1** -> Lecture de fifo[0] = 6
**CONSO 2** -> Lecture de fifo[1] = 7
**PROD** -> Production : fifo[2] = 8
**PROD** -> Production : fifo[0] = 9
**CONSO 1** -> Lecture de fifo[2] = 8
**CONSO 2** -> Lecture de fifo[0] = 9
**PROD** -> Production : fifo[1] = 10
**PROD** -> Production : fifo[2] = 11
**CONSO 1** -> Lecture de fifo[1] = 10
**CONSO 2** -> Lecture de fifo[2] = 11
**PROD** -> Production : fifo[0] = 12
**CONSO 1** -> Lecture de fifo[0] = 12
**PROD** -> Production : fifo[1] = 13
**CONSO 2** -> Lecture de fifo[1] = 13
**PROD** -> Production : fifo[2] = 14
**CONSO 1** -> Lecture de fifo[2] = 14
**PROD** -> Production : fifo[0] = 15
**CONSO 1** -> Lecture de fifo[0] = 15
**PROD** -> Production : fifo[1] = 16
**CONSO 1** -> Lecture de fifo[1] = 16
**PROD** -> Production : fifo[2] = 17
**CONSO 1** -> Lecture de fifo[2] = 17
**PROD** -> Production : fifo[0] = 18
**CONSO 1** -> Lecture de fifo[0] = 18
**PROD** -> Production : fifo[1] = 19
**CONSO 1** -> Lecture de fifo[1] = 19
**PROD** -> Production : fifo[2] = 20
**PROD** -> Production : fifo[0] = 21
**CONSO 1** -> Lecture de fifo[2] = 20
**CONSO 2** -> Lecture de fifo[0] = 21
**PROD** -> Production : fifo[1] = 22
**PROD** -> Production : fifo[2] = 23
**CONSO 1** -> Lecture de fifo[1] = 22
**CONSO 2** -> Lecture de fifo[2] = 23
**PROD** -> Production : fifo[0] = 24

```

FIGURE 2.8 – 2 consommateurs plus rapides que le producteur

2.5 2 Producteurs/ 2 Consommateurs

Finalement, nous avons implémenté le cas avec 2 producteurs et 2 consommateurs. Dans cet exemple, nous avons besoin des 2 mutex précédents, un pour empêcher les producteurs de produire en même temps et un autre pour empêcher les consommateurs de produire en même temps. Les producteurs partagent également des variables globales pour la file d'entiers et de même pour les consommateurs. Le principe reste exactement le même que précédemment.

```

1  #define TAILLE_TABLEAU 3
2
3  TACHE tacheStart();
4  TACHE tacheProd1();
5  TACHE tacheProd2();
6  TACHE tacheConso1();
7  TACHE tacheConso2();
8  uint16_t prod1, prod2, conso1, conso2;    // 2 conso et 2 prod
9  uint16_t fifo[TAILLE_TABLEAU];
10 uint16_t nb_places_libres = TAILLE_TABLEAU;
11 ushort sem1, sem2, semMutexConso, semMutexProd;
12
13 // variables globales pour les producteurs et consommateurs
14 int l = 0, j = 0, m = 0;
15
16 TACHE tacheStart(void)
17 {
18     puts("—————> EXEC tache Start");
19
20     prod1 = cree(tacheProd1);
21     prod2 = cree(tacheProd2);
22     conso1 = cree(tacheConso1);
23     conso2 = cree(tacheConso2);
24
25     active(conso1);
26     active(conso2);
27     active(prod1);
28     active(prod2);
29
30     fin_tache();
31 }
32
33 TACHE tacheProd1(void)
34 {
35     puts("—————> EXEC tache Prod 1");
36
37     uint16_t k;

```



```
40 while(1)
41 {
42     for (k=0; k<30000; k++);
43     //for (k=0; k<10000; k++);
44
45     s_wait(semMutexProd); // P du mutex (entrée de la zone critique)
46
47     if (nb_places_libres >= 1)
48     {
49         _lock_();
50         printf("**PROD 1** -> Production : fifo[%d] = %d\n", l, j);
51         fifo[l] = j;
52         _unlock_();
53         j++;
54
55         nb_places_libres--;
56
57         l++;
58         if (l == TAILLE_TABLEAU)
59             l = 0;
60     }
61     else
62     {
63         s_wait(sem1); // dort()
64     }
65
66     s_signal(semMutexProd); // V du mutex (sortie de la zone critique)
67
68     if (nb_places_libres < TAILLE_TABLEAU)
69     {
70         s_signal(sem2); // reveille(conso);
71     }
72 }
73
74
75 fin_tache();
76 }
77
78 TACHE tacheProd2(void)
79 {
80     puts("—————> EXEC tache Prod 2");
81
82     uint16_t k;
83
84     while(1)
85     {
86         for (k=0; k<30000; k++);
```

```

88     //for (k=0; k<10000; k++);
90     s_wait(semMutexProd); // P du mutex (entrée de la zone critique)
92     if (nb_places_libres >= 1)
93     {
94         _lock_();
95         printf(" **PROD 2** -> Production : fifo[%d] = %d\n", l, j);
96         fifo[l] = j;
97         _unlock_();
98         j++;
99
100        nb_places_libres--;
101
102        l++;
103        if (l == TAILLE_TABLEAU)
104            l = 0;
105    }
106    else
107    {
108        s_wait(sem1); // dort()
109    }
110
111    s_signal(semMutexProd); // V du mutex (sortie de la zone critique)
112
113    if (nb_places_libres < TAILLE_TABLEAU)
114    {
115        s_signal(sem2); // reveille(conso);
116    }
117
118    }
119
120    fin_tache();
121 }
122
123
124 TACHE tacheConso1(void)
125 {
126     uint16_t k;
127
128     puts("—————> EXEC tache Conso 1");
129
130     s_wait(sem2); // dort()
131
132     while(1)
133     {
134         //for (k=0; k<30000; k++);
135         for (k=0; k<10000; k++);
136

```

```
138     s_wait(semMutexConso); // P du mutex (entrée de la zone critique)
140
141     if (nb_places_libres==TAILLE_TABLEAU)
142     {
143         s_wait(sem2); // dort()
144     }
145     else
146     {
147         _lock_();
148         printf("**CONSO 1** -> Lecture de fifo[%d] = %d\n", m, fifo[m]);
149         _unlock_();
150
151         nb_places_libres++;
152
153         m++;
154         if (m==TAILLE_TABLEAU)
155             m=0;
156     }
157
158     s_signal(semMutexConso); // V du mutex (sortie de la zone critique)
159
160     if (nb_places_libres==1)
161     {
162         s_signal(sem1); // reveille(prod);
163     }
164 }
165
166 fin_tache();
167 }
168
169 TACHE tacheConso2(void)
170 {
171     uint16_t k;
172
173     puts("—————> EXEC tache Conso 2");
174
175     s_wait(sem2); // dort()
176
177     while(1)
178     {
179         //for (k=0; k<30000; k++);
180         for (k=0; k<10000; k++);
181
182         s_wait(semMutexConso); // P du mutex (entrée de la zone critique)
183
184     }
```

```

186     if (nb_places_libres==TAILLE_TABLEAU)
187     {
188         s_wait(sem2); // dort()
189     }
190     else
191     {
192         _lock_();
193         printf("**CONSO 2** -> Lecture de fifo[%d] = %d\n",m, fifo[m]);
194         _unlock_();
195
196         nb_places_libres++;
197
198         m++;
199         if (m==TAILLE_TABLEAU)
200             m=0;
201     }
202
203     s_signal(semMutexConso); // V du mutex (sortie de la zone critique)
204
205     if (nb_places_libres==1)
206     {
207         s_signal(sem1); // reveille(prod);
208     }
209 }
210
211 fin_tache();
212 }
213
214 int main()
215 {
216     serial_init(115200);
217     puts("Test noyau");
218     puts("Noyau preemptif");
219     puts("*****DEBUT*****\n\n\n\n\n");
220
221     printf("TAILLE DU TABLEAU : %d\n", TAILLE_TABLEAU);
222
223     s_init();
224     sem1 = s_cree(0); // prod
225     sem2 = s_cree(0); // conso
226
227     // mutex entre les 2 consommateurs pour les empêcher de consommer
228     // en même temps
229     semMutexConso = s_cree(1);
230
231     // mutex entre les 2 producteurs pour les empêcher de produire en

```

```
232     même temps  
    semMutexProd = s_cree(1);  
234     start(tacheStart);  
236     return(0);  
}
```

Pour cette exemple, nous avons testé trois configurations différentes :

- 1) Les producteurs et consommateurs ont tous la même vitesse (figure 2.9).
- 2) Les producteurs ont la même vitesse mais sont plus rapides que les consommateurs qui ont la même vitesse également (figure 2.10).
- 3) Les consommateurs ont la même vitesse mais sont plus rapides que les producteurs qui ont la même vitesse également (figure 2.11).

```

TAILLE DU TABLEAU : 3
-----> EXEC tache Start
-----> EXEC tache Conso 1
-----> EXEC tache Conso 2
-----> EXEC tache Prod 1
-----> EXEC tache Prod 2
**PROD 1** -> Production : fifo[0] = 0
**PROD 2** -> Production : fifo[1] = 1
**PROD 1** -> Production : fifo[2] = 2
**CONSO 1** -> Lecture de fifo[0] = 0
**CONSO 2** -> Lecture de fifo[1] = 1
**CONSO 1** -> Lecture de fifo[2] = 2
**PROD 1** -> Production : fifo[0] = 3
**PROD 2** -> Production : fifo[1] = 4
**CONSO 1** -> Lecture de fifo[0] = 3
**CONSO 1** -> Lecture de fifo[1] = 4
**PROD 1** -> Production : fifo[2] = 5
**CONSO 2** -> Lecture de fifo[2] = 5
**PROD 2** -> Production : fifo[0] = 6
**PROD 1** -> Production : fifo[1] = 7
**PROD 1** -> Production : fifo[2] = 8
**CONSO 2** -> Lecture de fifo[0] = 6
**CONSO 1** -> Lecture de fifo[1] = 7
**PROD 1** -> Production : fifo[0] = 9
**PROD 1** -> Production : fifo[1] = 10
**CONSO 2** -> Lecture de fifo[2] = 8
**PROD 2** -> Production : fifo[2] = 11
**CONSO 1** -> Lecture de fifo[0] = 9
**CONSO 2** -> Lecture de fifo[1] = 10
**CONSO 2** -> Lecture de fifo[2] = 11
**PROD 1** -> Production : fifo[0] = 12
**CONSO 2** -> Lecture de fifo[0] = 12
**PROD 2** -> Production : fifo[1] = 13
**CONSO 1** -> Lecture de fifo[1] = 13
**PROD 1** -> Production : fifo[2] = 14
**PROD 2** -> Production : fifo[0] = 15
**PROD 2** -> Production : fifo[1] = 16
**CONSO 2** -> Lecture de fifo[2] = 14
**PROD 2** -> Production : fifo[2] = 17
**CONSO 1** -> Lecture de fifo[0] = 15
**PROD 1** -> Production : fifo[0] = 18
**CONSO 2** -> Lecture de fifo[1] = 16
**CONSO 1** -> Lecture de fifo[2] = 17
**CONSO 1** -> Lecture de fifo[0] = 18
**PROD 2** -> Production : fifo[1] = 19
**CONSO 1** -> Lecture de fifo[1] = 19
**PROD 1** -> Production : fifo[2] = 20
**CONSO 2** -> Lecture de fifo[2] = 20
**PROD 2** -> Production : fifo[0] = 21
**PROD 1** -> Production : fifo[1] = 22
**PROD 1** -> Production : fifo[2] = 23
**CONSO 1** -> Lecture de fifo[0] = 21
**PROD 2** -> Production : fifo[0] = 24
**CONSO 2** -> Lecture de fifo[1] = 22
**PROD 1** -> Production : fifo[1] = 25
**CONSO 1** -> Lecture de fifo[2] = 23
**CONSO 1** -> Lecture de fifo[0] = 24
**PROD 2** -> Production : fifo[2] = 26
**CONSO 2** -> Lecture de fifo[1] = 25
**PROD 1** -> Production : fifo[0] = 27
**PROD 2** -> Production : fifo[1] = 28
**CONSO 1** -> Lecture de fifo[2] = 26
**PROD 2** -> Production : fifo[2] = 29
**CONSO 2** -> Lecture de fifo[0] = 27
**PROD 1** -> Production : fifo[0] = 30
**CONSO 1** -> Lecture de fifo[1] = 28
**CONSO 2** -> Lecture de fifo[2] = 29
**CONSO 2** -> Lecture de fifo[0] = 30
**PROD 2** -> Production : fifo[1] = 31

```

FIGURE 2.9 – 2 producteurs et 2 consommateurs à la même vitesse

```

TAILLE DU TABLEAU : 3
-----> EXEC tache Start
-----> EXEC tache Conso 1
-----> EXEC tache Conso 2
-----> EXEC tache Prod 1
-----> EXEC tache Prod 2
**PROD 1** -> Production : fifo[0] = 0
**PROD 2** -> Production : fifo[1] = 1
**PROD 1** -> Production : fifo[2] = 2
**CONSO 1** -> Lecture de fifo[0] = 0
**CONSO 2** -> Lecture de fifo[1] = 1
**CONSO 1** -> Lecture de fifo[2] = 2
**PROD 1** -> Production : fifo[0] = 3
**PROD 2** -> Production : fifo[1] = 4
**CONSO 2** -> Lecture de fifo[0] = 3
**PROD 1** -> Production : fifo[2] = 5
**CONSO 1** -> Lecture de fifo[1] = 4
**CONSO 2** -> Lecture de fifo[2] = 5
**PROD 2** -> Production : fifo[0] = 6
**PROD 1** -> Production : fifo[1] = 7
**CONSO 1** -> Lecture de fifo[0] = 6
**PROD 2** -> Production : fifo[2] = 8
**CONSO 2** -> Lecture de fifo[1] = 7
**CONSO 1** -> Lecture de fifo[2] = 8
**PROD 1** -> Production : fifo[0] = 9
**PROD 2** -> Production : fifo[1] = 10
**PROD 1** -> Production : fifo[2] = 11
**CONSO 2** -> Lecture de fifo[0] = 9
**CONSO 1** -> Lecture de fifo[1] = 10
**CONSO 2** -> Lecture de fifo[2] = 11
**PROD 2** -> Production : fifo[0] = 12
**PROD 1** -> Production : fifo[1] = 13
**CONSO 2** -> Lecture de fifo[0] = 12
**PROD 2** -> Production : fifo[2] = 14
**CONSO 1** -> Lecture de fifo[1] = 13
**PROD 1** -> Production : fifo[0] = 15
**CONSO 2** -> Lecture de fifo[2] = 14
**PROD 2** -> Production : fifo[1] = 16
**PROD 1** -> Production : fifo[2] = 17
**CONSO 2** -> Lecture de fifo[0] = 15
**PROD 2** -> Production : fifo[0] = 18
**CONSO 2** -> Lecture de fifo[1] = 16
**PROD 2** -> Production : fifo[1] = 19
**CONSO 1** -> Lecture de fifo[2] = 17
**PROD 2** -> Production : fifo[2] = 20
**CONSO 2** -> Lecture de fifo[0] = 18
**CONSO 1** -> Lecture de fifo[1] = 19
**CONSO 2** -> Lecture de fifo[2] = 20
**PROD 2** -> Production : fifo[0] = 21
**PROD 1** -> Production : fifo[1] = 22
**CONSO 2** -> Lecture de fifo[0] = 21
**PROD 2** -> Production : fifo[2] = 23
**PROD 1** -> Production : fifo[0] = 24
**CONSO 1** -> Lecture de fifo[1] = 22
**CONSO 2** -> Lecture de fifo[2] = 23
**CONSO 2** -> Lecture de fifo[0] = 24

```

FIGURE 2.10 – 2 producteurs plus rapides que les 2 consommateurs

```

TAILLE DU TABLEAU : 3
-----> EXEC tache Start
-----> EXEC tache Conso 1
-----> EXEC tache Conso 2
-----> EXEC tache Prod 1
-----> EXEC tache Prod 2
**PROD 2** -> Production : fifo[0] = 0
**CONSO 1** -> Lecture de fifo[0] = 0
**PROD 1** -> Production : fifo[1] = 1
**CONSO 1** -> Lecture de fifo[1] = 1
**PROD 2** -> Production : fifo[2] = 2
**PROD 1** -> Production : fifo[0] = 3
**PROD 2** -> Production : fifo[1] = 4
**CONSO 2** -> Lecture de fifo[2] = 2
**CONSO 1** -> Lecture de fifo[0] = 3
**PROD 1** -> Production : fifo[2] = 5
**CONSO 2** -> Lecture de fifo[1] = 4
**CONSO 1** -> Lecture de fifo[2] = 5
**PROD 2** -> Production : fifo[0] = 6
**PROD 1** -> Production : fifo[1] = 7
**PROD 2** -> Production : fifo[2] = 8
**CONSO 1** -> Lecture de fifo[0] = 6
**CONSO 2** -> Lecture de fifo[1] = 7
**PROD 2** -> Production : fifo[0] = 9
**CONSO 1** -> Lecture de fifo[2] = 8
**CONSO 2** -> Lecture de fifo[0] = 9
**PROD 1** -> Production : fifo[1] = 10
**CONSO 2** -> Lecture de fifo[1] = 10
**PROD 2** -> Production : fifo[2] = 11
**PROD 1** -> Production : fifo[0] = 12
**PROD 2** -> Production : fifo[1] = 13
**CONSO 2** -> Lecture de fifo[2] = 11
**CONSO 1** -> Lecture de fifo[0] = 12
**PROD 2** -> Production : fifo[2] = 14
**CONSO 2** -> Lecture de fifo[1] = 13
**CONSO 1** -> Lecture de fifo[2] = 14
**PROD 1** -> Production : fifo[0] = 15
**CONSO 1** -> Lecture de fifo[0] = 15
**PROD 2** -> Production : fifo[1] = 16
**PROD 1** -> Production : fifo[2] = 17
**PROD 2** -> Production : fifo[0] = 18
**CONSO 1** -> Lecture de fifo[1] = 16
**CONSO 2** -> Lecture de fifo[2] = 17
**PROD 2** -> Production : fifo[1] = 19
**CONSO 1** -> Lecture de fifo[0] = 18
**CONSO 2** -> Lecture de fifo[1] = 19
**PROD 1** -> Production : fifo[2] = 20
**CONSO 2** -> Lecture de fifo[2] = 20
**PROD 2** -> Production : fifo[0] = 21
**PROD 1** -> Production : fifo[1] = 22
**PROD 2** -> Production : fifo[2] = 23
**CONSO 2** -> Lecture de fifo[0] = 21

```

FIGURE 2.11 – 2 consommateurs plus rapides que les 2 producteurs

Rapport TP3 - Parties 5 et 6

5ème partie : Le dîner des philosophes

Le dîner des philosophes est un problème de synchronisation très connu. Cinq philosophes sont réunis autour d'une table circulaire pour dîner. Cinq couverts composés d'une assiette et d'une fourchette sont disposés autour de la table. Chaque philosophe a besoin de deux fourchettes pour manger son plat, la fourchette de gauche et la fourchette de droite. Un philosophe ne peut être que dans trois états, soit il pense, soit il a faim, soit il mange. On suppose dans notre cas que les fourchettes sont numérotées de 0 à 4 et que les philosophes sont numérotés de 1 à 5. Ainsi, le philosophe numéro 1 aura besoin de la fourchette 0 à sa gauche et de la fourchette 1 à sa droite, et le philosophe 5 aura besoin de la fourchette 4 à sa gauche et de la fourchette 0 à sa droite.

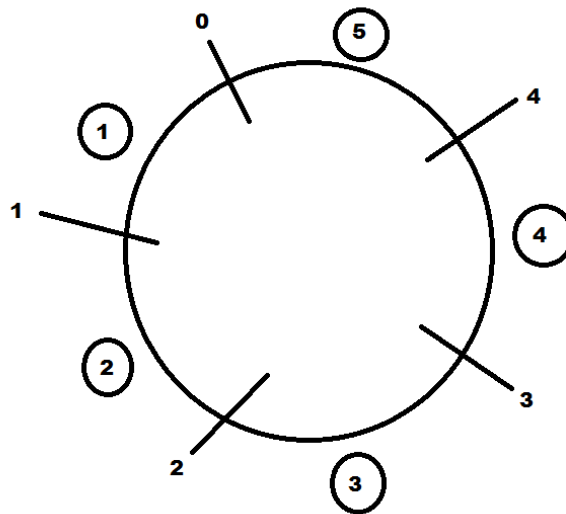


FIGURE 3.1 – La table des philosophes

Nous avons résolu ce problème à l'aide des sémaphores suivants :

- 1) Un tableau de 5 sémaphores initialisés à 1 pour l'accès aux fourchettes. Un sémaphore à 1 signifie que la fourchette est disponible et 0 qu'elle n'est pas disponible.
- 2) Un sémaphore initialisé à 4 qui garanti qu'une fourchette sera toujours disponible. En effet, normalement il n'est pas judicieux de mettre des sémaphores sur les fourchettes car ceci peut générer un interblocage si jamais les philosophes s'emparent tous de la fourchette gauche ou de la fourchette droite. C'est pour cela que nous avons ajouté un sémaphore qui autorise ou non la prise d'une fourchette. Comme ce sémaphore est initialisé à 4, il autorise au maximum la prise de 4 fourchettes donc il restera toujours une fourchette de disponible.
- 3) Un sémaphore d'attente initialisé à 0 qui permet à chaque philosophe qui a fini de manger d'attendre la fin des autres. Lorsqu'un philosophe fini de manger, il se bloque sur ce sémaphore. Ceci permet de finir proprement le repas. On suppose ici que chaque philosophe n'a qu'un plat et qu'il n'y a pas de second repas.
- 4) Un sémaphore de type mutex initialisé à 1 qui permet de protéger la variable qui compte le nombre de philosophes ayant fini de manger.

En résumé, un philosophe pense puis lorsqu'il a faim il demande au sémaphore numéro 2 s'il peut prendre une fourchette. Si oui, c'est à dire s'il reste au moins deux fourchettes, alors il demande au tableau de sémaphores numéro 1 s'il peut s'emparer de la fourchette de gauche et de droite. La fourchette de gauche associé à ce philosophe se trouve à l'indice $(id - 1) \% 5$ avec id l'identifiant du philosophe et la fourchette de droite se trouve à l'indice $id \% 5$. Par exemple, le philosophe 5 tentera de s'emparer de la fourchette $(5 - 1) \% 5 = 4 \% 5 = 4$ à sa gauche et de la fourchette $id \% 5 = 5 \% 5 = 0$ à sa droite. Si le philosophe a pris ses deux fourchettes, alors il mange et lorsqu'il a terminé, il libère le sémaphore numéro 2, c'est à dire qu'il incrémente le nombre de fourchettes que tous les philosophes peuvent prendre en même temps. Il libère aussi les sémaphores associés aux fourchettes qu'il vient d'utiliser via le tableau de sémaphores numéro 1. Puis il demande au mutex numéro 4 s'il peut incrémenter la variable qui compte le nombre de philosophes ayant fini de manger. Si oui, alors il y a deux cas possibles, soit la variable est inférieure à 5 ce qui veut dire que des philosophes n'ont pas encore fini de manger, soit elle vaut 5 ce qui veut dire que tout le monde a fini de manger. Dans le premier cas, le philosophe se bloque sur le sémaphore d'attente numéro 3 pour attendre ses collègues et dans le deuxième cas, tous les philosophes sont libérés et c'est la fin du repas.

```
1 TACHE tacheStart();
  TACHE tachePhilo();
3
  // semProtectFouch => garanti qu'une fourchette sera toujours
  // disponible
5 // semAttente => permet à chaque philosophe qui a fini de manger d'
  // attendre la fin des autres
  // mutex => protéger la variable qui compte le nombre de philosophes
  // ayant fini de manger
7 ushort semProtectFouch, semAttente, mutex;
9
  // tableau de sémaphores des fourchettes
  ushort semFourchette[5];
11
  ushort id_philo; // id unique par philosophe
13
  // variable qui compte le nombre de philosophes ayant fini de manger
15 int j;
17 TACHE tacheStart(void)
  {
19     puts("————> EXEC tache Start");
21
    int i;
    s_init(); // initialisation des sémaphores
23
    // autorise au maximum la prise de 4 fourchettes
25     semProtectFouch = s_cree(4);
27
    mutex = s_cree(1);
    semAttente = s_cree(0);
29
    for(i=0; i < 5; i++)
31         semFourchette[i] = s_cree(1);
33
    for(id_philo=0; id_philo < 5; id_philo++)
        active(cree(tachePhilo));
35
    fin_tache();
37 }
39 TACHE tachePhilo()
  {
41     uint16_t k;
43
    ushort id = id_philo; // récupération de l'id du philosophe
    printf("————> Philosophe n° : %d demarre\n", id); // il pense
45
```

```
47 while (1)
48 {
49     for (k=0; k<60000; k++); // il pense
50
51     _lock_();
52     printf("————> Philosophe n° : %d attend pour manger : il a faim\n", id);
53     _unlock_();
54
55     // autoriser à prendre une fourchette ?
56     s_wait(semProtectFourch);
57
58     // on prend la fourchette de gauche
59     s_wait(semFourchette[ (id - 1) % 5 ]);
60
61     // on prend la fourchette de droite
62     s_wait(semFourchette[ id % 5 ]);
63
64     _lock_();
65     printf("————> Philosophe n° : %d en train de manger\n", id);
66     _unlock_();
67
68     for (k=0; k<30000; k++); // temps d'attente pour qu'il mange
69
70     _lock_();
71     printf("————> Philosophe n° : %d a fini de manger : il pense\n", id);
72     _unlock_();
73
74     // autorisation de prendre une fourchette de plus
75     s_signal(semProtectFourch);
76
77     // on depose la fourchette de gauche
78     s_signal(semFourchette[ (id - 1) % 5 ]);
79
80     // on depose la fourchette de droite
81     s_signal(semFourchette[ id % 5 ]);
82
83     s_wait(mutex); // pour protéger le compteur j
84     j++; // un philosophe de plus a fini de manger
85
86     if(j == 5) // ils ont tous fini de manger
87     {
88         for(k=0; k < 5; k++)
89             s_signal(semAttente); // on libere toutes les tâches
90
91         j = 0;
92         s_signal(mutex);
93     }
```

```

93     printf("————> Fin du repas\n", id);
95     fin_tache();
97 }
98 else
99 {
100     s_signal(mutex);
101     s_wait(semAttente); // on bloque la tâche, le philosophe attend
102
103     fin_tache(); // fin du repas
104 }
105 }
106 }
107 }
108
109 int main()
110 {
111     serial_init(115200);
112     puts("Test noyau");
113     puts("Noyau preemptif");
114     puts("*****DEBUT*****\n\n\n\n\n");
115
116     start(tacheStart);
117
118     return(0);
119 }

```

```

-----> EXEC tache Start
-----> Philosophe n° : 1 démarre
-----> Philosophe n° : 2 démarre
-----> Philosophe n° : 3 démarre
-----> Philosophe n° : 4 démarre
-----> Philosophe n° : 5 démarre
-----> Philosophe n° : 2 attend pour manger : il a faim
-----> Philosophe n° : 2 en train de manger
-----> Philosophe n° : 1 attend pour manger : il a faim
-----> Philosophe n° : 3 attend pour manger : il a faim
-----> Philosophe n° : 4 attend pour manger : il a faim
-----> Philosophe n° : 4 en train de manger
-----> Philosophe n° : 2 a fini de manger : il pense
-----> Philosophe n° : 4 a fini de manger : il pense
-----> Philosophe n° : 1 en train de manger
-----> Philosophe n° : 5 attend pour manger : il a faim
-----> Philosophe n° : 3 en train de manger
-----> Philosophe n° : 3 a fini de manger : il pense
-----> Philosophe n° : 1 a fini de manger : il pense
-----> Philosophe n° : 5 en train de manger
-----> Philosophe n° : 5 a fini de manger : il pense
-----> Fin du repas

```

FIGURE 3.2 – Le dîner des philosophes

On voit bien sur la figure précédente que lorsque le philosophe 2 mange, les philosophes 1 et 3 sont logiquement bloqués. Seul le philosophe 4 est autorisé à manger en même que le philosophe 2. De même, lorsque le philosophe 1 mange, seul le philosophe 3 peut manger en même temps sachant que les philosophes 2 et 4 ont fini de manger et que le philosophe 5 est bloqué. Enfin, lorsque le philosophe 5 a fini de manger, le repas se termine.

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* :

```

1 #ifndef PIPE_H
2 #define PIPE_H
3
4 #include "noyau.h"
5
6 #define MAX_PIPES 5 //Nombre de tubes
7 #define SIZE_PIPE 10 //Taille de chaque tube
8
9 typedef struct
10 {
11     ushort pr_w , pr_r ; // redacteur & lecteur du tube
12     ushort ocupp ; // donnees restantes
13     uchar is , ie ; // pointeurs d'entree / sortie
14     uchar tube[SIZE_PIPE] ; // Tampon
15 } PIPE;
16
17 PIPE __pipe[MAX_PIPES] ; // Variables tubes
18
19 //Allocation du conduit
20 unsigned int p_open(unsigned int redacteur , unsigned int lecteur);
21 //Libération du tube
22 void p_close (unsigned int conduit);
23 //Lecture dans un tube
24 void p_read (int tube , uchar* donnees , int quantite);
25 //Ecriture dans un tube
26 void p_write(int tube , uchar* donnees , int quantite);
27 //Initialisation des tubes
28 void init_pipes();
29 
```

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

2.1 Fonction `init_pipes`

```
1 void init_pipes()
2 {
3     unsigned int i;
4     for (i=0; i<MAX_PIPES; i++)
5         _pipe[i].pr_w=MAX_TACHES; //Tube inutilisé
6 }
```

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
2 unsigned int p_open(unsigned int redacteur, unsigned int lecteur)
3 {
4     //Vérifier si les tâches sont créées
5     if (_contexte[redacteur].status == NCREER || _contexte[lecteur].
6         status == NCREER)
7         return -1;
8
9     //Vérifier qu'il n'existe pas de tube avec ces 2 tâches
10    unsigned int i;
11    for (i=0 ; i<MAX_PIPES ; i++)
12    {
13        if (_pipe[i].pr_w == redacteur && _pipe[i].pr_r == lecteur)
14            //Il existe un tube avec ces 2 tâches
15            return -1;
16    }
17
18    //Trouver un tube non utilisé
19    i=0;
20    while ( (_pipe[i].pr_w != MAX_TACHES) && (i < MAX_PIPES) )
21        i++;
22    if (i == MAX_PIPES) //Aucun tube n'est libre
23        return -2;
24
25    //Initialisation du tube
26    _pipe[i].pr_w = redacteur;
27    _pipe[i].pr_r = lecteur;
28    _pipe[i].is = _pipe[i].ie = 0;
29    _pipe[i].ocupp = 0; //Données restantes (0 au départ)
30
31    //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 fermer 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 tâche  
  |         if (__pipe[tube].occup == 0)  
17 |         {  
  |             printf("Tampon vide ==> endormissement de la tâche %d\n", __pipe  
  | [tube].pr_r);
```



```

19     dort();
20 }
21
22     donnees[i] = _pipe[tube].tube[_pipe[tube].is];
23     _pipe[tube].is++;
24     _pipe[tube].ocupp--;
25
26     //Si le tube était plein et si
27     //la tâche écrivain est suspendue sur une
28     //écriture dans ce tube alors la réveiller
29     if ( (_pipe[tube].ocupp+1 == SIZE_PIPE) && (_contexte[_pipe[tube]
30     ].pr_w).status == SUSP) )
31         reveille(_pipe[tube].pr_w);
32 }
33 }

```

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 (à l'emplacement de sortie "is"). On incrémente 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 tentative 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
2 void p_write(int tube, uchar* donnees, int quantite)
3 {
4     //Vérifier que le tube existe et que la tâche en est propriétaire
5     if ( (_pipe[tube].pr_w != _tache_c) )
6     {
7         printf("Tache non autorisee a ecrire dans le pipe\n");
8     }
9 }

```

```

8      return;
9  }
10
11  //Copie des données dans le tube
12  int i;
13  for (i=0 ; i < quantite ; i++)
14  {
15      //Vérifier qu'il y a de la place dans le tampon, sinon endormir
16      la tache
17      if (_pipe[tube].ocupp == SIZE_PIPE)
18      {
19          printf("Plus de place dans le tampon ==> endormissement de la
20          tache %d\n", _pipe[tube].pr_w);
21          dort();
22      }
23
24      //Copie des données
25      _pipe[tube].tube[_pipe[tube].ie] = donnees[i];
26      _pipe[tube].ie++;
27      _pipe[tube].ocupp++;
28
29      //Si le tube était vide et si
30      //la tache lectrice est suspendue sur
31      //une lecture de ce tube alors la réveiller
32      if ( (_pipe[tube].ocupp-1 == 0) && (_contexte[_pipe[tube].pr_r].
33      status == SUSP) )
34          reveille(_pipe[tube].pr_r);
35  }
36 }

```

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 tâche 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.

2.6 Test de ces fonctions

Pour tester notre système de tubes, on réalise un programme de deux tâches : la tâche A et la tâche B. Deux tubes leur sont associés : un de A vers B et l'autre de B vers A. Le déroulement du programme est le suivant :

- La tâche A envoie le message "salut" à la tâche B via le premier tube
- La tâche B reçoit ce message via ce même tube
- La tâche B envoie ce message reçu à la tâche A via le deuxième tube
- La tâche A reçoit le message via le deuxième tube.

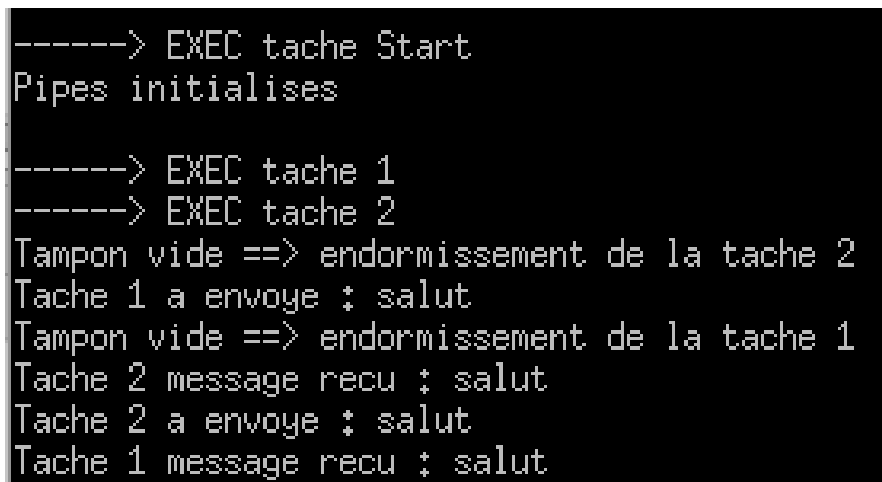
Voici le code associé :

```
1 #include "pipe.h"
2 #include "noyau.h"
3 #define TAILLE_MESSAGE 10
4
5 //definitions des fonctions citées précédemment
6
7 TACHE tacheStart();
8 TACHE tacheA();
9 TACHE tacheB();
10 uint16_t tA, tB;
11 int pipe1, pipe2;
12
13 TACHE tacheStart(void)
14 {
15     puts("————> EXEC tache Start");
16
17     init_pipes();
18     puts("Pipes initialises\n");
19
20     tA = cree(tacheA);
21     tB = cree(tacheB);
22
23     pipe1 = p_open(tA, tB); // pipe de A vers B
24     pipe2 = p_open(tB, tA); // pipe de B vers A
25
26     active(tA);
27     active(tB);
28
29
30     fin_tache();
31 }
32
33 TACHE tacheA()
```

```
34 {  
    puts("————> EXEC tache 1");  
36  
    uint16_t k;  
38    uchar envoiA[TAILLE_MESSAGE] = "salut";  
    uchar recuA[TAILLE_MESSAGE] = "";  
40  
    for (k=0; k<30000; k++);  
42  
    p_write(pipe1, envoiA, 5);  
44  
    __lock__();  
46    printf("Tache 1 a envoye : %s\n", envoiA);  
    __unlock__();  
48  
    p_read(pipe2, recuA, 5);  
50  
    __lock__();  
52    printf("Tache 1 message recu : %s\n", recuA);  
    __unlock__();  
54  
    fin_tache();  
56 }  
58  
TACHE tacheB()  
60 {  
    puts("————> EXEC tache 2");  
62  
    uint16_t k;  
64    uchar recuB[TAILLE_MESSAGE] = "";  
66  
    for (k=0; k<30000; k++);  
68  
    p_read(pipe1, recuB, 5);  
70  
    __lock__();  
    printf("Tache 2 message recu : %s\n", recuB);  
72    __unlock__();  
74  
    p_write(pipe2, recuB, 5);  
76  
    __lock__();  
    printf("Tache 2 a envoye : %s\n", recuB);  
78    __unlock__();  
80  
    fin_tache();  
82 }
```

```
84 int main()  
86 {  
    serial_init(115200);  
88    puts("*****DEBUT PIPE*****\n\n");  
90    start(tacheStart);  
}
```

Le résultat obtenu est le suivant :



```
-----> EXEC tache Start  
Pipes initialises  
  
-----> EXEC tache 1  
-----> EXEC tache 2  
Tampon vide ==> endormissement de la tache 2  
Tache 1 a envoye : salut  
Tampon vide ==> endormissement de la tache 1  
Tache 2 message reçu : salut  
Tache 2 a envoye : salut  
Tache 1 message reçu : salut
```

FIGURE 3.3 – Résultat du pipe

On constate que notre système de tubes fonctionne bien. La tâche B reçoit bien le message de A et est capable de lui renvoyer. La tâche A reçoit bien le retour de B.