

# TP MI11 - Réalisation d'un mini noyau temps réel ARM

Théophile DANCOISNE et Louis FRERET

Juin 2017

Au cours de ce TP, nous avons réalisé en C un mini noyau temps réel pour une carte ARM ARMADÉUS.

## 1 Ordonnanceur de tâches

Pour commencer notre noyau, nous avons implémenté un ordonnanceur de tâche simple dont nous allons présenter le code. Le rôle de l'ordonnanceur est de gérer l'activité des tâches prêtes ou en exécution.

Rappel : Le contexte d'un processus correspond à une image des registres à un instant  $t$ . En commutant la valeur du pointeur du registre du processeur, on effectue un changement de contexte.

```
1  /* NOYAUFILE.C */
2  /*-----*/
3  /*  gestion de la file d'attente des taches pretes et actives          */
4  /*  la file est rangee dans un tableau. ce fichier decrit toutes      */
5  /*  les primitives de base                                           */
6  /*-----*/
7
8  #include "serialio.h"
9  #include "noyau.h"
10
11 /* variables communes a toutes les procedures */
12 /*-----*/
13
14 static uint16_t _file[MAX_TACHES]; /* indice=numero de tache */
15 /* valeur=tache suivante */
16 static uint16_t _queue; /* valeur de la derniere tache */
17 /* pointe la prochaine tache a activer */
18
19 /*      initialisation de la file      */
20 /*-----*/
21 entre : sans
22 sortie : sans
23 description : la queue est initialisee vide, queue prend la valeur de tache
24               impossible
25 */
26
27 void file_init(void) {
28     _queue = F_VIDE;
29     for (int i = 0; i < MAX_TACHES; i++) {
30         _file[i] = F_VIDE;
31     }
32 }
33
34 /*      ajouter une tache dans la pile      */
35 /*-----*/
36 entre : n numero de la tache a entrer
37 sortie : sans
38 description : ajoute la tache n en fin de pile
39 */
40
41 void ajoute(uint16_t n) {
42     printf("ajoute(%d)\n", n);
43 }
```

```

44  if (_queue != F_VIDE && _file[n] != F_VIDE) {
45  printf("Error: Tâche déjà existante.\n");
46  return;
47  }
48  if (_queue == F_VIDE) {
49  _file[n] = n;
50  } else {
51  _file[n] = _file[_queue];
52  _file[_queue] = n;
53  }
54  _queue = n;
55  }
56
57  uint16_t predecesseur(uint16_t t);
58  /*          retire une tâche de la file          */
59  *-----*
60  entree : t numero de la tâche a sortir
61  sortie : sans
62  description: sort la tâche t de la file. L'ordre de la file n'est pas
63  modifie
64  */
65
66  void retire(uint16_t t) {
67  printf("retire(%d)\n", t);
68  if (_file[t] == F_VIDE) {
69  printf("Error: Tâche inexistante.\n");
70  }
71
72  if (_queue == t) {
73  _queue = _file[t];
74  }
75
76  uint16_t pred_t = predecesseur(t);
77  _file[pred_t] = _file[t];
78
79  _file[t] = F_VIDE;
80  }
81
82  uint16_t predecesseur(uint16_t t) {
83  uint16_t pred_t;
84  for (int i = 0; i < MAX_TACHES; i++) {
85  if (_file[i] == t) {
86  return i;
87  }
88  }
89  }
90
91  /*          recherche du suivant a executer          */
92  *-----*
93  entree : sans
94  sortie : t numero de la tâche a activer
95  description : la tâche a activer est sortie de la file. queue pointe la
96  suivante
97  */
98  uint16_t suivant(void) {
99  if (_queue == F_VIDE) {
100  printf("Error: Aucune tâche.");
101  return F_VIDE;
102  } else {
103  uint16_t suivant = _file[_queue];
104  _queue = suivant;
105  return suivant;
106  }
107  }
108
109  /*          affichage du dernier element          */
110  *-----*
111  entree : sans

```

```

112 sortie : sans
113 description : affiche la valeur de queue
114 */
115
116 void affic_queue(void) {
117     printf("Affichage de la queue:\n");
118
119     char *format = "Queue:\t%d\tValeur:\t%d\n";
120     printf(format, _queue, _file[_queue]);
121 }
122
123 /*      affichage de la file      */
124 *-----*
125 entree : sans
126 sortie : sans
127 description : affiche les valeurs de la file
128 */
129
130 void affic_file(void) {
131     printf("Affichage de la file:\n");
132
133     char *format = "Indice:\t%d\tValeur:\t%d\n";
134     for (int i = 0; i < MAX_TACHES; i++) {
135         printf(format, i, _file[i]);
136     }
137 }

```

Listing 1 – noyau.c

L'ordonnancement ainsi implémenté est le plus simple du monde : chaque tâche est exécutée tour à tour sans priorité en respectant l'ordre défini par l'utilisateur.

Commentons les points importants de cet ordonnanceur. L'ajout d'une tâche dans la pile se déroule de la manière suivante. Si la file est vide, ce qui se manifeste lorsque la queue est égale au nombre maximum de tâches, la nouvelle tâche sera évidemment à la fois son successeur et son prédécesseur. Sinon le successeur de la nouvelle tâche aura pour valeur le successeur de la queue. Le successeur de la queue sera la nouvelle tâche. Le nouvelle queue prend pour valeur la nouvelle tâche. Retirer une tâche s'effectue de la même manière. Il s'agit de rattaché le prédécesseur de la tâche à retirer avec le successeur de la tâche à retirer. Et bien sûr, on veille à mettre à jour la valeur de la queue si c'est la dernière tâche.

Nous avons ensuite testé cet ordonnanceur en reproduisant l'exemple donné par l'énoncé. L'affichage que nous avons obtenu correspond bien à celui attendu.

```

1 // Ce programme a pour but de tester les différentes fonctions de NOYAUFIL.C
2 // de la partie 1 de "Réalisation dun mini noyau temps réel ARM"
3
4 #include <noyau.h>
5 #include <serialio.h>
6
7 int main(int argc, char **argv) {
8     file_init();
9
10    ajoute(3);
11    ajoute(5);
12    ajoute(1);
13    ajoute(0);
14    ajoute(2);
15
16    affic_file();
17    affic_queue();
18
19    uint16_t tache_suivante = suivant();
20    printf("Tâche suivante:\t%d\n", tache_suivante);
21
22    affic_file();
23    affic_queue();
24
25    retire(0);
26

```

```
27  affic_file();
28  affic_queue();
29
30  ajoute(6);
31
32  affic_file();
33  affic_queue();
34
35  return 0;
36 }
```

Listing 2 – testfile.c

## 2 Gestion et commutation de tâches

A chaque fois que l'on effectue un changement de contexte, il faut désactiver les interruptions pour éviter

1. de perdre le contexte du processus en cours d'exécution
2. d'entrer dans un état incohérent où 2 processus auraient l'état running.

Nous allons à présent décrire les différentes fonctions de l'ordonnanceur, implémentées dans le fichier *noyau.c*. Des primitives dépendant du matériel préalablement définies dans le fichier *noyau.h* ont été utilisées.

### 2.1 Sortie du noyau

Pour cette procédure, il suffit de désactiver les interruptions en faisant appel à la primitive `_irq_disable_` puis éventuellement d'afficher le nombre d'exécutions de chaque tâche. La sortie du noyau déclenche une boucle infinie pour éviter de perdre le contrôle du processeur.

### 2.2 Destruction d'une tâche

La destruction d'une tâche s'effectue en 3 étapes après avoir préalablement désactivé les interruptions :

- Changer l'état de la tâche à celui de CREE, c'est-à-dire connue du noyau avec une pile allouée et un identifiant.
- Sortir la tâche de la file
- Appeler le gestionnaire de tâches (*sheduler*).

### 2.3 Créer une nouvelle tâche

Le rôle de cette fonction est "d'allouer un espace dans la pile à la tâche et lui attribue un identifiant, qui est retourné à l'appelant".

Les opération de création du contexte, d'allocation d'une pile et décrémentation du pointeur de pile pour la nouvelle tâche doivent bien entendu être effectués en section critique. Les primitives utilisées à cet effet sont `_lock_` et `_unlock_`. Si le nombre de tâche est maximal, alors on sort du noyau pour cause de dépassement. Le contexte de la nouvelle tâche est récupéré via le tableau des contextes. L'initialisation du contexte de la tâche se réalise en affectant l'adresse du sommet de la pile, auquel on accède par la variable `_tos`, au pointeur de pile initial du contexte (`sp_init`). Il faut alors bien entendu mettre à jour l'adresse du sommet de la pile en le décrémentant de `PILE_TACHE + PILE_IRQ`, c'est-à-dire la taille max de la pile d'une tâche plus la taille max de la pile IRQ par tâche. Enfin, il faut mémoriser l'adresse du début de la tâche et changer son état à CREEER.

### 2.4 Activer une tâche

Cette fonction place la tâche dans la file des tâches prêtes. Un test préalable doit être effectué pour vérifier que la tâche a bien été créée. Si ce n'est pas le cas, on sort du noyau. Si la tâche est bien à l'état CREE, après être entré en section critique, on effectue les 3 opérations suivantes :

- Changer le statut de la tâche à "prêt".
- Ajouter la tâche dans la liste.

— Activer une tâche prête en faisant appel au scheduler.

## 2.5 Appel au gestionnaire de tâches

L'appel au gestionnaire de tâches s'effectue par l'appel à la fonction *schedule*, qui effectuera un branchement (un saut en assembleur) sur *scheduler*. Bien entendu, toute la procédure doit s'exécuter en section critique. Le flag indiquant qu'il faut acquitter le timer est paramétré à "faux". On entre alors en mode IRQ à l'aide de la primitive *\_set\_arm\_mode\_*. Le branchement est alors fait sur le cœur de l'ordonnancement. Une fois son traitement accompli, on repasse en mode système avec *\_set\_arm\_mode\_* puis l'on sort de la section critique. Décrivons précisément la commutation de contexte. La première étape consiste à mémoriser le contexte de la tâche en cours en sauvegardant le pointeur de pile dans le champ *sq\_irq*. Via la fonction *suivant*, définie précédemment dans *noyaufil.c*, on récupère le numéro de la tâche suivante à exécuter. S'il n'y a plus rien à ordonnancer, on sort du noyau. On incrémente alors le compteur d'activation de la tâche suivante, indiquant le nombre de fois où l'on a commuté sur la tâche. Puis si son statut est à prêt, on charge son pointeur de pile initial *sp\_ini* et l'on passe en mode système. Via l'opération dans *sp\_ini - PILE\_IRQ*, on charge le pointeur de pile courant en mode système. On change alors le statut de la tâche à EXEC, c'est-à-dire en possession du processeur. On autorise alors les interruptions et on lance la tâche. On restaure alors le contexte complet depuis la pile IRQ.

## 2.6 Démarrer le noyau et commencer une première tâche

Cette fonction "initialise les structures de données du noyau, met en place le gestionnaire d'interruption scheduler" et "crée et active la première tâche, dont l'adresse est passée en paramètres". La première étape consiste à initialiser l'état des tâches à NCREE. On initialise ensuite la tâche courante ainsi que la file, par un appel à *file\_init*. L'adresse du sommet de la pile, *\_tos* est ensuite initialisée à la valeur de *sp*, c'est-à-dire de la pile. En mode IRQ, on paramètre *sp\_irq* à *\_tos*; et l'on repasse en mode système. Après avoir désactivé les interruptions, on initialise le timer, de type *imx\_timer* à 100Hz : *tcmp*, la fréquence de l'ordonnanceur, est ainsi paramétrée à 10000. On crée alors une première tâche et on l'active.

## 3 Exclusion mutuelle

Les différentes tâches peuvent partager des ressources. Afin de gérer l'accès aux ressources et éviter d'éventuels interblocages, nous allons implémenter dans le noyau des mécanismes d'exclusion mutuelle permettant aux programmes de s'approprier une ressource et d'en interdire l'accès à toute autre tâche. Les premiers mécanismes que nous allons implémenter concernent l'attente passive. L'attente passive, contrairement à l'attente active, ne consomme pas de ressources processeur.

### 3.1 Endormir la tâche courante

Pour endormir une tâche d'une telle manière, nous allons la retirer de la file d'attente du scheduler. Avant cela, nous entrons dans une section critique à l'aide de la primitive *lock()*. L'état de la tâche devient *SUSP* pour les différencier avec les tâches en cours d'exécution (*EXEC*). Enfin on lance un appel à *schedule()* pour forcer un changement de tâche immédiatement dès que tous les appels de *\_unlock\_()* seront dépilés.

```

1 void dort(void) {
2     // Entrée en section critique
3     _lock_();
4
5     // Changement de statut de la tâche
6     _contexte[_tache_c].status = SUSP;
7
8     // On retire la tâche de la liste du scheduler
9     retire(_tache_c);
10
11     // On interrompt la tâche actuelle pour passer à la suivante
12     schedule();
13 }
```

```

14 // Fin de section critique
15 _unlock_();
16 }

```

Listing 3 – Dort()

### 3.2 Réveille une tâche

Cette fonction réveille une tâche. Elle ajoute ainsi la tâche à la liste du scheduler si et seulement si la tâche est suspendue. Le statut de la tâche redevient bien évidemment l'état *EXEC*.

```

1 void reveille(uint16_t t) {
2     // Récupération du contexte associée à la tâche
3     CONTEXTE *p = &_contexte[t];
4
5     // Si la tâche est suspendue
6     if (p->status == SUSP)
7     {
8         // Entrée en section critique
9         _lock_();
10
11        // La tâche est en exécution
12        p->status = EXEC;
13
14        // La tâche est ajoutée dans la liste du scheduler
15        ajoute(t);
16
17        // Sortie de section critique
18        _unlock_();
19    }
20 }

```

Listing 4 – Reveille()

### 3.3 Programme Producteur/Consommateur

Afin de tester les fonctions écrites ci-dessus, nous écrivons un simple programme producteur/consommateur. Les deux tâches partagent la même ressource : une FIFO circulaire que l'un remplit et l'autre vide. Il faut bien évidemment endormir le consommateur si il y a famine. Mais il faut également endormir le producteur si la FIFO est pleine. Bien entendu, le producteur va réveiller le consommateur quand des données seront à consommer dans la FIFO. Inversement, le consommateur va réveiller le producteur si la FIFO n'est plus remplie. L'inconvénient de cette méthode est que les appels à *dort()* et *reveille()* sont à gérer "manuellement" et augmenter le nombre de producteurs et consommateurs complexifie grandement le problème.

## 4 Les sémaphores

Les sémaphores sont des outils logiciels, inventés par Dijkstra, permettant de synchroniser des tâches et à partager des ressources. Le sémaphore est initialisé avec une valeur positive ou nulle. Lorsqu'une tâche tente d'accéder à la ressource, elle décrémente la valeur du sémaphore. Si cette valeur est nulle, alors la ressource est bloquée et la tâche doit attendre d'être réveillée. Lorsqu'une tâche libère une ressource, alors elle incrémente la valeur du sémaphore, ce qui débloque immédiatement l'accès à un processus bloqué.

### 4.1 Implémentation du sémaphore

Dans sa structure, le sémaphore doit gérer la file des processus qui sont bloqués pour ainsi réveiller les tâches automatiquement lorsque la ressource devient à nouveau disponible.

```

1 // Structure du sémaphore
2 typedef struct {
3     FIFO file;          /* File circulaire des tâches en attente */
4     short valeur;       /* compteur du sémaphore e(s) */

```

```

5  short occupp;    /* si le semaphore est libre */
6  } SEMAPHORE;

```

Listing 5 – sem.h

Voici l'implémentation des fonction *s\_wait()* et *s\_signal*. Ces fonctions nécessitent d'être placées dans une section critique du noyau afin d'éviter tout changement de contexte pendant un accès au sémaphore.

```

1  // Requête d'accès au sémaphore
2  void s_wait(short n)
3  {
4      _lock_();
5
6      // Se bloque sur le sémaphore ou décrémente la valeur
7      if(_sem[n].valeur <= 0)
8      {
9          push_fifo(&_sem[n].file, _tache_c);
10         dort();
11     }
12     else
13     {
14         _sem[n].valeur = _sem[n].valeur - 1;
15     }
16     _unlock_();
17 }
18
19 // Libération du sémaphore
20 void s_signal(short n)
21 {
22     _lock_();
23
24     // Incrémente la valeur
25     // Ou réveille une tâche bloquée
26
27     int i, wakeup = 0;
28     for (i = 0; i < MAX_FIFO; ++i)
29     {
30         if (empty_fifo(&_sem[n].file) != 1)
31         {
32             wakeup = 1;
33             int t = top_fifo(&_sem[n].file);
34             pop_fifo(&_sem[n].file);
35             reveille(t);
36         }
37     }
38     if(wakeup == 0) _sem[n].valeur = _sem[n].valeur + 1;
39     _unlock_();
40 }

```

Listing 6 – sem.c

Notez bien que dans cette implémentation, lorsqu'une tâche libère le sémaphore alors que d'autres sont bloquées, la valeur du sémaphore n'est pas incrémentée puisque une tâche débloquée ne décrémente pas le sémaphore non plus.

## 4.2 Programme Producteur/Consommateur

Le programme du producteur/consommateur peut dès lors être écrit plus simplement à l'aide des sémaphores. Les appels aux fonctions *dort()* et *reveille()* sont gérés par le sémaphore. Nous devons créer deux sémaphores. Le premier comptera le nombre de places libres dans la FIFO partagée. Le second sémaphore comptera le nombre d'éléments dans la FIFO. Ainsi, un producteur commencera par décrémente la valeur du premier sémaphore, ajoutera un élément à la FIFO, puis incrémentera la valeur du second sémaphore. Le consommateur quant à lui commencera par décrémente le second sémaphore, retirera un élément de la FIFO, puis incrémentera la valeur du premier sémaphore. Avec ce système, nous pouvons créer autant de producteurs et de consommateurs qu'on souhaite sans se soucier des attentes des tâches puisque celles-ci sont gérées par le mécanisme de sémaphores.

## 5 Le dîner des philosophes

Le problème bien connu du dîner des philosophes nous a permis de mettre à profit notre travail réalisé pour mettre en place les sémaphores (voir 4).

Nous avons utilisé les structures de données suivantes pour représenter le problème :

```

1 typedef struct {
2     short mutex; // index du sémaphore
3 } fourchette;
4
5 typedef struct {
6     fourchette *main_droite;
7     fourchette *main_gauche;
8 } philosophe;
9
10 philosophe *philosophes[NB_PHILOSOPHES];
11 fourchette *fourchettes[NB_PHILOSOPHES];

```

Listing 7 – "Le dîner des philosophes - structures"

Une fourchette est simplement représentée par un mutex, c'est-à-dire un sémaphore agissant comme une porte fermée si une tâche est en section critique, ouverte sinon. Chaque philosophe garde l'adresse de deux fourchettes, une dites "à gauche", l'autre "à droite".

Le comportement d'un philosophe est défini de la manière suivante :

```

1 void comportement_philosophe(int id_philosophe) {
2     printf("====> EXEC tache philosophe_%d\n", id_philosophe);
3
4     int nb_repas_necessaire = 30;
5
6     int id_gauche = id_philosophe == NB_PHILOSOPHES - 1 ? 0 : id_philosophe + 1;
7     int id_droit = id_philosophe == 0 ? NB_PHILOSOPHES : id_philosophe - 1;
8
9     fourchette *fourchette_gauche = &fourchettes[id_philosophe];
10    fourchette *fourchette_droite = &fourchettes[id_droit];
11
12    philosophe* philosophe_gauche = &philosophes[id_gauche];
13    philosophe* philosophe_droit = &philosophes[id_droit];
14
15    philosophe* moi = &philosophes[id_philosophe];
16
17    int j;
18    while (nb_repas_necessaire > 0) {
19        nb_repas_necessaire--;
20
21        printf("====> Philosophe %d pense.\n", id_philosophe);
22        for (j = 0; j < 30000L; j++);
23
24        printf("====> Philosophe %d est affame.\n", id_philosophe);
25        /* chaque philosophe partage sa fourchette gauche
26         * avec le philosophe de gauche pour qui c'est la fourchette droite */
27        s_wait(philosophe_gauche->main_droite->mutex);
28        s_wait(philosophe_droit->main_gauche->mutex);
29
30        printf("====> Philosophe %d mange.\n", id_philosophe);
31        for (j = 0; j < 30000L; j++);
32        s_signal(fourchette_gauche->mutex);
33        s_signal(fourchette_droite->mutex);
34    }
35
36    printf("====> Philosophe %d ne mangera plus.\n", id_philosophe);
37 }

```

Listing 8 – "Le dîner des philosophes - comportement d'un philosophe"

A chaque fois qu'un philosophe veut manger, il appelle la primitive `s_wait`, autrement dit la primitive P, sur le mutex de sa fourchette gauche et le mutex de sa fourchette droite. Une fois entrée dans les deux sections critiques associées à la fourchette gauche et à la fourchette droite, la tâche associée au philosophe peut lancer



l'activité de "manger". Une fois cette activité terminée, la primitive V est appelée par cette tâche, via la fonction `s_signal` pour libérer les deux mutex.

## 6 Communication par tubes

Les tubes sont des moyens de communication entre deux tâches. Le tube est exclusivement réservé aux deux tâches indiquées lors de sa création. Le tube est un canal de communication unidirectionnel. L'une des deux tâches peut écrire dans le tube tandis que l'autre tâche peut lire les données sur le tube.

### 6.1 Structure de données

La structure du tube ressemble à celle d'une FIFO circulaire. Plusieurs informations supplémentaires viennent cependant compléter la structure dont notamment les tâches propriétaires du tube et un booléen afin de savoir si une tâche est endormie sur le tube.

```

1 typedef struct {
2     short pr_w , pr_r ;      /* redacteur & lecteur du tube */
3     short sleep_w, sleep_r; /* indicateur pour savoir si dort à cause du pipe */
4     short occup;             /* donnees restantes */
5     int size;                /* donnees dans le pipe */
6     char is , ie;            /* pointeurs dentree / sortie */
7     char tube[SIZE_PIPE];    /* Tampon */
8 } PIPE;

```

Listing 9 – "Structure du pipe"

### 6.2 Lecture et écriture

La FIFO circulaire est de taille fixe dans le noyau. Il se peut alors que la tâche qui écrit des données remplisse la file auquel cas elle s'endormira. De la même manière, si la tâche qui lit les données n'a plus de données à lire, elle s'endormira jusqu'à ce que de nouvelles données soient à consommer. C'est le pipe qui implémente les mécanismes qui endorment et réveillent les tâches.

```

1 // Ecriture bloquante d'une donnée dans le pipe
2 void p_write(unsigned conduit , char* donnees , unsigned nb)
3 {
4     // Si le pipe est initialisé et que la tâche courante est bien rédactrice
5     if(_pipe[conduit].occup == 1 && _pipe[conduit].pr_w == _tache_c)
6     {
7         // On traite les données une par une
8         int i;
9         for(i = 0; i < nb; ++i)
10        {
11            // Si le pipe est plein
12            if(_pipe[conduit].size == SIZE_PIPE)
13            {
14                // Ecriture bloquante
15                _pipe[conduit].sleep_w = 1;
16                dort();
17            }
18
19            // Ecriture des données
20            _lock();
21            _pipe[conduit].tube[_pipe[conduit].ie] = donnees[i];
22            _pipe[conduit].ie = (_pipe[conduit].ie + 1) % SIZE_PIPE;
23            _pipe[conduit].size = _pipe[conduit].size + 1;
24            _unlock();
25
26            // Si la tâche lectrice est endormie sur une lecture et que le pipe n'est pas vide
27            if(_pipe[conduit].size != 0 && _pipe[conduit].sleep_r == 1)
28            {
29                // Réveiller la tâche lectrice
30                _pipe[conduit].sleep_r = 0;

```

```

31     reveille(_pipe[conduit].pr_r);
32 }
33 }
34 }
35 }

```

Listing 10 – "Ecriture dans un tube"

Il est possible d'écrire plusieurs données dans le pipe lors du même appel à la fonction write. La fonction itérative copie les données une par une dans le pipe. Si le pipe est plein, la tâche s'endort. Si après une écriture, il s'avère que le lecteur est endormi sur le pipe, alors on le réveille. La fonction de lecture des données fonctionne de manière analogue.

```

1 // Lecture bloquante d'une donnée dans le pipe
2 void p_read (unsigned conduit, char* donnees, unsigned nb)
3 {
4     // Si le pipe est initialisé et que la tâche courante est bien lecteur
5     if(_pipe[conduit].ocupp == 1 && _pipe[conduit].pr_r == _tache_c)
6     {
7         // On traite les données une par une
8         int i;
9         for(i = 0; i < nb; ++i)
10        {
11            // Si il n'y a rien à lire
12            if(_pipe[conduit].size == 0)
13            {
14                // Lecture bloquée
15                _pipe[conduit].sleep_r = 1;
16                dort();
17            }
18
19            // Lecture des données
20            _lock_();
21            donnees[i] = _pipe[conduit].tube[_pipe[conduit].is];
22            _pipe[conduit].is = (_pipe[conduit].is + 1) % SIZE_PIPE;
23            _pipe[conduit].size = _pipe[conduit].size - 1;
24            _unlock_();
25
26            // Si la tâche rédactrice dort et que le pipe n'est pas plein
27            if(_pipe[conduit].size != SIZE_PIPE && _pipe[conduit].sleep_w == 1)
28            {
29                // Réveiller la tâche
30                _pipe[conduit].sleep_w = 0;
31                reveille(_pipe[conduit].pr_w);
32            }
33        }
34    }
35 }

```

Listing 11 – "Ecriture dans un tube"