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 ARMADEUS.

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 est correspond à une image des registres du processus à un instant t. En commutant la valeur du pointeur du registre du processeur, on effectue un changement de contexte.

```
/* NOYAUFILE.C */
/*-------------------------------------------------------------------------*
 *  gestion de la file d'attente des taches pretes et actives              *
 *  la file est rangee dans un tableau. ce fichier decrit toutes           *
 *  les primitives de base                                                 *
 *-------------------------------------------------------------------------*/

#include "serialio.h"
#include "noyau.h"

/* variables communes a toutes les procedures *
 *--------------------------------------------*/

static uint16_t _file[MAX_TACHES];    /* indice=numero de tache */
/* valeur=tache suivante  */
static uint16_t _queue;               /* valeur de la derniere tache */
/* pointe la prochaine tache a activer */

/*     initialisation de la file      *
 *------------------------------------*
entre  : sans
sortie : sans
description : la queue est initialisee vide, queue prend la valeur de tache
        impossible
*/

void file_init(void) {
  _queue = F_VIDE;
  for (int i = 0; i < MAX_TACHES; i++) {
    _file[i] = F_VIDE;
  }
}

/*        ajouter une tache dans la pile      *
 *--------------------------------------------*
entree : n numero de la tache a entrer
sortie : sans
description : ajoute la tache n en fin de pile
*/

void ajoute(uint16_t n) {
  printf("ajoute(%d)\n", n);

```

```
  if (_queue != F_VIDE && _file[n] != F_VIDE) {
  printf("Error: Tâche déjà existante.\n");
  return;
  }
  if (_queue == F_VIDE) {
    _file[n] = n;
  } else {
    _file[n] = _file[_queue];
    _file[_queue] = n;
  }
  _queue = n;
}

uint16_t predecesseur(uint16_t t);
/*           retire une tache de la file        *
 *----------------------------------------------*
entree : t numero de la tache a sortir
sortie : sans
description: sort la tache t de la file. L'ordre de la file n'est pas
       modifie
*/

void retire(uint16_t t) {
  printf("retire(%d)\n", t);
  if (_file[t] == F_VIDE) {
    printf("Error: Tâche inexistante.\n");
  }

  if(_queue == t) {
    _queue = _file[t];
  }

  uint16_t pred_t = predecesseur(t);
  _file[pred_t] = _file[t];

  _file[t] = F_VIDE;
}

uint16_t predecesseur(uint16_t t) {
  uint16_t pred_t;
  for (int i = 0; i < MAX_TACHES; i++) {
    if (_file[i] == t) {
    return i;
    }
  }
}

/*        recherche du suivant a executer       *
 *----------------------------------------------*
entree : sans
sortie : t numero de la tache a activer
description : la tache a activer est sortie de la file. queue pointe la
        suivante
*/
uint16_t suivant(void) {
  if (_queue == F_VIDE) {
    printf("Error: Aucune tâche.");
    return F_VIDE;
  } else {
  uint16_t suivant = _file[_queue];
  _queue = suivant;
    return suivant;
  }
}

/*     affichage du dernier element     *
 *--------------------------------------*
entree : sans
```

```
sortie : sans
description : affiche la valeur de queue
*/

void affic_queue(void) {
  printf("Affichage de la queue:\n");

  char *format = "Queue:\t%d\tValeur:\t%d\n";
  printf(format, _queue, _file[_queue]);
}

/*     affichage de la file     *
 *------------------------------*
entree : sans
sortie : sans
description : affiche les valeurs de la file
*/

void affic_file(void) {
  printf("Affichage de la file:\n");

  char *format = "Indice:\t%d\tValeur:\t%d\n";
  for (int i = 0; i < MAX_TACHES; i++) {
    printf(format, i, _file[i]);
  }
}
```

Listing 1 – noyaufil.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 prédecesseur. 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 ratâcher le prédecesseur de la tâche à retirer avec le sucesseur 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 ordonanceur en reproduisant l'exemple donné par l'énoncé. L'affichage que nous avons obtenu correspond bien à celui attendu.

```
// Ce programme a pour but de tester les différentes fonctions de NOYAUFIL.C
// de la partie 1 de "Réalisation dun mini noyau temps réel ARM"

#include <noyau.h>
#include <serialio.h>

int main(int argc, char **argv) {
  file_init();

  ajoute(3);
  ajoute(5);
  ajoute(1);
  ajoute(0);
  ajoute(2);

  affic_file();
  affic_queue();

  uint16_t tache_suivante = suivant();
  printf("Tâche suivante:\t%d\n", tache_suivante);

  affic_file();
  affic_queue();

  retire(0);

```

```
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'exécution 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 désactiver 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 un 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 CREER.

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 a 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 accomplie, on replace en mode système avec $\_set\_arm\_mode\_$ puis l'on sort de la section critique. Décrivons précisément la communication 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 on l'on a commuté sur la tâche. Puis si son statut est à prêt, on charge sont 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 en l'initialisation de l'état des tâches, en paramétrant 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 créé alors une première tâche et on l'active.

Résumé

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.

3 Exclusion mutuelle

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. Une autre fonction sera quand à elle capable de réveiller une tâche et de la

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
```

```
    // On interromp la tâche actuelle pour passer à la suivante
    schedule();

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

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*.

```
void reveille(uint16_t t) {
    // Récupération du contexte associée à la tâche
    CONTEXTE *p = &_contexte[t];

    // Si la tâche est suspendue
    if (p->status == SUSP)
    {
        // Entrée en section critique
        _lock_();

        // La tâche est en exécution
        p->status = EXEC;

        // La tâche est ajoutée dans la liste du scheduler
        ajoute(t);

        // Sortie de section critique
        _unlock_();
    }
}
```

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 Djikstra, 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.

```c
// Structure du sémaphore
typedef struct {
  FIFO file;      /* File circulaire des tâches en attente */
  short valeur ;    /* compteur du sémaphore e(s) */
  short ocupp;    /* si le semaphore est libre */
} 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.

```c
// Requête d'accès au sémaphore
void s_wait(short n)
{
  _lock_();

  // Se bloque sur le sémaphore ou décrémente la valeur
  if(_sem[n].valeur <= 0)
  {
    push_fifo(&_sem[n].file, _tache_c);
    dort();
  }
  else
  {
    _sem[n].valeur = _sem[n].valeur - 1;
  }
  _unlock_();
}

// Libération du sémaphore
void s_signal(short n)
{
  _lock_();

  // Incrémente la valeur
  // Ou réveille une tâche bloquée

  int i, wokeup = 0;
  for (i = 0; i < MAX_FIFO; ++i)
  {
    if (empty_fifo(&_sem[n].file) != 1)
    {
      wokeup = 1;
      int t = top_fifo(&_sem[n].file);
      pop_fifo(&_sem[n].file);
      reveille(t);
    }
  }
  if(wokeup == 0) _sem[n].valeur = _sem[n].valeur + 1;
  _unlock_();
}
```

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 au 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 décrémenter 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 décrémenter 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 sémaphore (voir 4).

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

```
typedef struct {
  short mutex; // index du sémaphore
} fourchette;

typedef struct {
  fourchette *main_droite;
  fourchette *main_gauche;
} philosophe;

philosophe *philosophes[NB_PHILOSOPHES];
fourchette *fourchettes[NB_PHILOSOPHES];
```

Listing 7 – "Le diner 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 :

```
void comportement_philosophe(int id_philosophe) {
  printf("------> EXEC tache philosophe_%d\n", id_philosophe);

  int nb_repas_necessaire = 30;

  int id_gauche = id_philosophe == NB_PHILOSOPHES - 1 ? 0 : id_philosophe + 1;
  int id_droit = id_philosophe == 0 ? NB_PHILOSOPHES : id_philosophe - 1;

  fourchette *fourchette_gauche = &fourchettes[id_philosophe];
  fourchette *fourchette_droite = &fourchettes[id_droit];

  philosophe* philosophe_gauche = &philosophes[id_gauche];
  philosophe* philosophe_droit = &philosophes[id_droit];

  philosophe* moi = &philosophes[id_philosophe];

  int j;
  while (nb_repas_necessaire > 0) {
    nb_repas_necessaire--;

    printf("======> Philosophe %d pense.\n", id_philosophe);
    for (j = 0; j < 30000L; j++);

    printf("======> Philosophe %d est affame.\n", id_philosophe);
    /* chaque philosophe partage sa fourchette gauche
     * avec le philosophe de gauche pour qui c'est la fourchette droite */
    s_wait(philosophe_gauche->main_droite->mutex);
    s_wait(philosophe_droit->main_gauche->mutex);

    printf("======> Philosophe %d mange.\n", id_philosophe);
    for (j = 0; j < 30000L; j++);
    s_signal(fourchette_gauche->mutex);
    s_signal(fourchette_droite->mutex);
  }

  printf("======> Philosophe %d ne mangera plus.\n", id_philosophe);
```

```
}
```

Listing 8 – "Le diner 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 appellée 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.

```
typedef struct {
  short pr_w , pr_r ;      /* redacteur & lecteur du tube */
  short sleep_w, sleep_r;   /* indicateur pour savoir si dort à cause du pipe */
  short ocupp;          /* donnees restantes */
  int size;           /* donnes dans le pipe */
  char is, ie;          /* pointeurs dentree / sortie */
  char tube[SIZE_PIPE];    /* Tampon */
} 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.

```
// Ecriture bloquante d'une donnée dans le pipe
void p_write(unsigned conduit, char* donnees, unsigned nb)
{
  // Si le pipe est initialisé et que la tâche courante est bien rédactrice
  if(_pipe[conduit].ocupp == 1 && _pipe[conduit].pr_w == _tache_c)
  {
    // On traite les données une par une
    int i;
    for(i = 0; i < nb; ++i)
    {
      // Si le pipe est plein
      if(_pipe[conduit].size == SIZE_PIPE)
      {
        // Ecriture bloquante
        _pipe[conduit].sleep_w = 1;
        dort();
      }

      // Ecriture des données
      _lock_();
      _pipe[conduit].tube[_pipe[conduit].ie] = donnees[i];
      _pipe[conduit].ie = (_pipe[conduit].ie + 1) % SIZE_PIPE;
```

```
      _pipe[conduit].size = _pipe[conduit].size + 1;
      _unlock_();

      // Si la tâche lectrice est endormie sur une lecture et que le pipe n'est pas vide
      if(_pipe[conduit].size != 0 && _pipe[conduit].sleep_r == 1)
      {
        // Réveiller la tâche lectrice
        _pipe[conduit].sleep_r = 0;
        reveille(_pipe[conduit].pr_r);
      }
    }
  }
}
```

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.

```
// Lecture bloquante d'une donnée dans le pipe
void p_read (unsigned conduit, char* donnees, unsigned nb)
{
  // Si le pipe est initialisé et que la tâche courante est bien lecteur
  if(_pipe[conduit].ocupp == 1 && _pipe[conduit].pr_r == _tache_c)
  {
    // On traite les données une par une
    int i;
    for(i = 0; i < nb; ++i)
    {
      // Si il n'y a rien à lire
      if(_pipe[conduit].size == 0)
      {
        // Lecture bloquée
        _pipe[conduit].sleep_r = 1;
        dort();
      }

      // Lecture des données
      _lock_();
      donnees[i] = _pipe[conduit].tube[_pipe[conduit].is];
      _pipe[conduit].is = (_pipe[conduit].is + 1) % SIZE_PIPE;
      _pipe[conduit].size = _pipe[conduit].size - 1;
      _unlock_();

      // Si la tâche rédactrice dort et que le pipe n'est pas plein
      if(_pipe[conduit].size != SIZE_PIPE && _pipe[conduit].sleep_w == 1)
      {
        // Réveiller la tâche
        _pipe[conduit].sleep_w = 0;
        reveille(_pipe[conduit].pr_w);
      }
    }
  }
}
```

Listing 11 – "Ecriture dans un tube"