TP MI11 - Réalisation d'un mini noyau temps réel ARM - Parties 1 et 2

Théophile DANCOISNE et Louis FRERET

Mai 2017

1 Ordonnanceur de tâches

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) {
  if (_queue == F_VIDE) {
    _file[n] = n;
  }

  if (_file[n] == F_VIDE) {
    _file[n] = suivant();
```

```
48      _file[_queue] = n;
49    } else {
50      printf("Error: Tâche déjà existante.");
51    }
52
53  }
54
55  uint16_t predecesseur(uint16_t t);
56  /*           retire une tache de la file        *
57   *-----------------------------------------------*
58  entree : t numero de la tache a sortir
59  sortie : sans
60  description: sort la tache t de la file. L'ordre de la file n'est pas
61         modifie
62  */
63
64  void retire(uint16_t t) {
65    if (_file[t] == F_VIDE) {
66      printf("Error: Tâche inexistante.");
67    }
68
69    uint16_t pred_t = predecesseur(t);
70    _file[pred_t] = _file[t];
71
72    _file[t] = F_VIDE;
73  }
74
75  uint16_t predecesseur(uint16_t t) {
76    uint16_t pred_t;
77    for (int i = 0; i < MAX_TACHES; i++) {
78      if (_file[i] == t) {
79        pred_t = i;
80        break;
81      }
82    }
83    return pred_t;
84  }
85
86  /*        recherche du suivant a executer       *
87   *-----------------------------------------------*
88  entree : sans
89  sortie : t numero de la tache a activer
90  description : la tache a activer est sortie de la file. queue pointe la
91          suivante
92  */
93  uint16_t suivant(void) {
94    if (_queue == F_VIDE) {
95      printf("Error: Aucune tâche.");
96      return F_VIDE;
97    } else {
98      return _file[_queue];
99    }
100 }
101
102 /*     affichage du dernier element     *
103  *--------------------------------------*
104 entree : sans
105 sortie : sans
106 description : affiche la valeur de queue
107 */
108
109 void affic_queue(void) {
110   printf("Affichage de la queue:\n");
111
112   char *format = "Queue:\t%d\tValeur:\t%d\n";
113   printf(format, _queue, _file[_queue]);
114 }
115
```

```
116 /*      affichage de la file      *
117  *--------------------------------*
118 entree : sans
119 sortie : sans
120 description : affiche les valeurs de la file
121 */
122
123 void affic_file(void) {
124   printf("Affichage de la file:\n");
125
126   char *format = "Indice:\t%d\tValeur:\t%d\n";
127   for (int i = 0; i < MAX_TACHES; i++) {
128     printf(format, i, _file[i]);
129   }
130 }
```

Listing 1 – noyaufil.c

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

```
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'exécution de chaque tâche.

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

Listing 3 – noyau.c

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*).

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

Listing 4 – noyau.c

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.

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


    /* debut section critique */
    _lock_();
```

```
    /* numero de tache suivant */
    tache++;

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

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

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


    /* fin section critique */
    _unlock_();

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

Listing 5 – noyau.c

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.

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

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

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

Listing 6 – noyau.c

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.

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

    /* Sauvegarder le contexte complet sur la pile IRQ */

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

    /* Reinitialiser le timer si necessaire */
    if (_ack_timer) {
        register struct imx_timer *tim1 = (struct imx_timer *) TIMER1_BASE;
        tim1->tstat &= ~TSTAT_COMP;
    } else {
        _ack_timer = 1;
    }

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

    /* tache prete ? */
    if (p->status == PRET) {
        /* Charger sp_irq initial */
        sp = p->sp_ini;
        /* Passer en mode syst?me */
        _set_arm_mode_(ARMMODE_SYS);
        /* Charger sp_sys initial */
```

```
        sp = p->sp_ini - PILE_IRQ;
        /* status tache -> execution */
        p->status = EXEC;
        /* autoriser les interuptions   */
        _irq_enable_();
        /* lancement de la tache */
        /* lancement de la tache */
        (*p->tache_adr)();
    } else {
        /* tache deja en execution, restaurer sp_irq */
        sp = p->sp_irq;
    }

    /* Restaurer le contexte complet depuis la pile IRQ */

    __asm__ __volatile__(
    /* Restaurer lr_irq */
            "ldmfd  sp!, {r0, lr}\t\n"
            /* et spsr_irq */
            "msr    spsr, r0\t\n"
            /* Restaurer registres mode system */
            "ldmfd  sp, {r0-r14}^\t\n"
            /* Attendre un cycle */
            "nop\t\n"
            /* Ajuster pointeur de pile irq */
            "add    sp, sp, #60\t\n"
            /* Retour d'exception */
            "subs   pc, lr, #4\t\n");
}
void schedule(void) {
    /* Debut section critique */
    _lock_();

    /* On simule une exception irq pour forcer un appel correct a scheduler().*/

    _ack_timer = 0;
    /* Passer en mode IRQ */
    _set_arm_mode_(ARMMODE_IRQ);
    __asm__ __volatile__(
    /* Sauvegarder cpsr dans spsr */
    "mrs  r0, cpsr\t\n"
            "msr  spsr, r0\t\n"
            /* Sauvegarder pc dans lr et l'ajuster */
            "add  lr, pc, #4\t\n"
            /* Saut Ã  scheduler */
            "b    scheduler\t\n"
    );
    /* Repasser en mode system */
    _set_arm_mode_(ARMMODE_SYS);

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

Listing 7 – noyau.c

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.

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

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

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

    /* on interdit les interruptions */
    _irq_disable_();

    /* Initialisation du timer Ã  100 Hz */

    tim1->tcmp = 10000;
    tim1->tprer = 0;
    tim1->tctl |= TCTL_TEN | TCTL_IRQEN | TCTL_CLKSOURCE_PERCLK16;

    /* Initialisation de l'AITC */

    aitc->intennum = TIMER1_INT;

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

Listing 8 – noyau.c

2.7 Endormir la tâche courante

Le rôle de cette fonction est d'endormir la tâche courante et d'attribuer le processeur à la suivante.

2.8 Réveille une tâche

Cette fonction réveille une tâche.