

# Compte-rendu TP "Réalisation d'un mini noyau temps réel – Partie 1 et 2"

Mewen Michel et Sander Ricou – MI11 UTC

Le but de ce TP est de comprendre un noyau temps réel simple et de réaliser un ordonnanceur de tâches fonctionnant en harmonie avec le premier.

## 1ère partie : Ordonnanceur de tâches

Etudions le fichier `noyaufil.c`. Nous devons compléter les fonctions vides suivantes :

- `file_init()` : initialise la file de tâches
- `ajoute(uint16 n)` : ajoute la tâche `n` en fin de file
- `suivant()` : retourne la tâche à activer
- `retire(uint16 n)` : retire la tâche `n` de la file sans modifier l'ordre.

Dans le code, `_queue` est le numéro de la tâche actuellement active. `_file[n]` nous donne la tâche suivante de la tâche `n`.

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

Pour initialiser la file, on attribue à chaque case et à la `_queue` une valeur de tâche impossible, `F_VIDE`. Cela nous permettra par la suite de reconnaître cette valeur.

```
void    ajoute ( uint16_t n )
{
    if(_queue != F_VIDE) {
        uint16_t tmp = _file[_queue];
        _file[_queue] = n;
        _file[n] = tmp;
    } else {
        _file[n] = n;
    }
    _queue = n;
}
```

Deux cas se présentent quand on souhaite ajouter une tâche à la file actuelle :

- soit la queue n'est pas initialisée : on ajoute donc la tâche au tableau en l'indiquant elle même en tant que tâche suivante et on fait pointer `_queue` sur `n`.
- soit la queue est déjà initialisée : on garde `tmp`, la prochaine tâche à exécuter. La tâche qui suivra la tâche courante sera donc `n` (la nouvelle tâche) et la tâche qui suivra la tâche `n` est donc `tmp`. Finalement, on définit la tâche courante `_queue` à `n` pour que la tâche suivante

soit correcte (ie. qu'on exécute bien tmp au prochain appel à suivant()).

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

On prend la valeur de la prochaine tâche à exécuter, on l'enregistre dans \_queue et on la retourne.

```
void    retire( uint16_t t )
{
    int tmp;
    for(int i = 0; i < MAX_TACHES; i++) {
        if(_file[i] == t){
            tmp = i;
            break;
        }
    }
    _file[tmp] = _file[t];
    _file[t] = F_VIDE;
}
```

On trouve tmp, la tâche qui précède t et on lui donne pour tâche suivante la tâche suivant t. On oublie pas de mettre la valeur de \_file[t] à vide.

Les deux fonctions suivantes permettent d'afficher le contenu de la file afin de la débbugger.

```
void affic_queue( void )
{
    printf("Tache active / queue = %d", _queue);
}

void affic_file( void )
{
    if(_queue == F_VIDE){
        print("File vide");
        return;
    }

    int i = _queue;

    do {
        print("%d -> %d", i, _file[i]);
        i = _file[i];
    } while (i != _queue);
}
```

Nous avons donc écrit un programme de test, testfil.c, reprenant l'exemple du sujet :

```
#include <stdint.h>
#include "serialio.h"
#include "noyau.h"

void main() {
    file_init();

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

    affic_file();
    affic_queue();

    suivant();

    affic_file();
    affic_queue();

    retire(0);

    affic_file();
    affic_queue();

    ajoute(6);

    affic_file();
    affic_queue();
}
```

Qui nous donne le résultat –satisfaisant– suivant :

```
2 -> 3
3 -> 5
5 -> 1
1 -> 0
0 -> 2
Tache active / queue = 2
3 -> 5
5 -> 1
1 -> 0
0 -> 2
2 -> 3
Tache active / queue = 3
3 -> 5
5 -> 1
1 -> 2
2 -> 3
Tache active / queue = 3
6 -> 5
5 -> 1
1 -> 2
2 -> 3
3 -> 6
Tache active / queue = 6
```

## 2ème partie : gestion et commutation de tâches

Nous allons dans cette partie expliquer le fonctionnement du noyau temps réel simple de `noyau.c`.

```

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

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

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

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

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

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

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

```

On appelle `start()` pour démarrer notre noyau avec une tâche départ. `adr_tache` est l'adresse à laquelle se trouve la fonction de la tâche de départ. Dans l'ordre, on va donc :

- récupérer le pointeur de pile
- pour chaque tâche dans la tableau contexte, initialiser leur etat à NCRE car aucune tâche n'est créée au départ
- initialiser la tâche courante à zéro : certains noyaux TR ont une tâche de fond mais ici il n'y en a pas... Ici, par défaut, la première tâche avec laquelle on active le noyau est positionnée en 0 des différents tableaux utilisés dans le code (`_contexte`, compteurs d'activations). De plus, quand on crée une tâche (voir la fonction `cree()` ci-dessous) l'identifiant généré tâche commence bien à 0 puis est incrémenté de 1 à chaque nouvelle création.
- initialiser de notre file avec la fonction `file_init()` créée précédemment
- initialiser le haut de la pile des tâches à la valeur courante du pointeur de pile
- passer temporairement en mode IRQ pour initialiser le pointeur de pile à une valeur correcte pour quand on aura une exception
- appeler `_irq_disable()` pour invalider les interruptions normales dans le Current Program Status Register. Les interruptions seront à nouveau activées dans le

scheduler().

- initialiser le timer chargé des changements de contexte. Lorsqu'il arrivera à son terme, il va créer une interruption IRQ *via* l'AITC
- paramétrer l'ARM Interrupt Controller pour qu'il retransmette les interruptions gérées par le timer
- finalement, on va créer et lancer la tâche de départ.

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

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

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

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

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

    _unlock();                  /* fin section critique */

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

cree() affecte un numéro à chaque tâche entre 0 (initialisation à -1 et ++ ) et MAX\_TACHES (qui vaut ici 8). Si on essaie de créer plus de tâches que possible, le noyau se met en erreur (noyau\_exit()). On aloue également une pile (de longueur PILE\_TACHE + PILE\_IRQ) à la tâche sur le haut de la pile actuelle (\_tos). On enregistre évidemment l'adresse à laquelle se trouve la fonction à réaliser par la tâches (p->tache\_adr) et on met son statut à CREE (donc en attente d'activation, selon l'énoncé). Ces opérations sont bien sûr réalisées dans une section critique car une préemption entre \_lock() et \_unlock() laisserait le noyau dans un état instable. \_lock() désactive les interruptions de la même manière qu'\_irq\_disable() et enregistre en plus le CPSR sur la pile et \_unlock() restaure cette valeur, réactivant par là même les interruptions.

```

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

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

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

```

active() prépare une tâche préalablement créée pour qu'elle puisse être ordonnancée, donc exécutée. En pratique, cette fonction passe le statut d'une tâche CREE à PRET, met la tâche dans la file et demande un ordonnancement. Si la tâche n'a pas été créée (NCREE, la valeur d'initialisation des tâches du tableau \_contexte), le noyau se met en défaut.

```

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();
}

```

La fonction fin\_tache() fait l'inverse de active() : on enlève la tâche de la file et on lui retire son status PRET (... à être exécuté). Vu qu'on a plus rien à exécuter dans cette tâche, on tente d'exécuter la tâche suivante.

```

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

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

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

```

schedule() fait un appel au gestionnaire de tâches scheduler avec les bonnes préparations. On commence par mettre \_ack\_timer à 0, ce qui permet d'indiquer une interruption logicielle au scheduler (pour ne pas réinitialiser le timer). On fait ensuite le changement de mode, les sauvegardes nécessaires, l'appel au scheduler pour revenir ensuite au mode initial.

```

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

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

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

    _contexte[_tache_c].sp_irq = sp; /* memoriser le pointeur de pile */
    _tache_c = suivant();             /* recherche du suivant */
    if (_tache_c == F_VIDE)
    {
        printf("Plus rien a ordonnancer.\n");
        noyau_exit();                /* Sortie du noyau */
    }
}

```



```

}
compteurs[_tache_c]++;          /* Incrémenter le compteur d'activations */
p = &contexte[_tache_c];       /* p pointe sur la nouvelle tache courante*/

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

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

```

Le scheduler() est le gestionnaire d'interruptions (déclaré comme tel dans le vecteur d'exception du fichier de démarrage ASM). Il peut être appelé par une interruption matérielle (timer qui échoue) ou logicielle (appel à scheduler()). Dans l'ordre, on va donc :

- sauvegarder l'état actuel (contexte et pointeur de pile)
- réinitialiser le timer en cas d'interruption matérielle
- appeler l'ordonnanceur pour connaître la tâche à exécuter maintenant
- s'arrêter s'il n'y a plus de tâche en état d'être exécutée dans la file
- créer/restituer l'état de la tâche (les commentaires donnés semblent suffisamment explicites)

```

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

```

noyau\_exit() est la fonction permettant de "terminer" le noyau. En réalité, nos processeurs étant incapable de ne rien faire, on lui fait exécuter une boucle infinie (for(;;)).

Voici le code commenté de notre fichier `noyautest.c` expérimentant, sur la cible, le noyau TR :

```
/* NOYAUTEST.C */
/*-----*
 *               Programme de tests               *
 *-----*/

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

/*
 ** Test du noyau preemptif. Lier noyautest.c avec noyau.c et noyauhil.c
 */

TACHE    tacheA(void);
TACHE    tacheB(void);
TACHE    tacheC(void);
TACHE    tacheD(void);

TACHE    tacheA(void)
{
    puts("-----> EXEC tache A");
    active(cree(tacheB));
    active(cree(tacheC));
    active(cree(tacheD));
    fin_tache();
}

TACHE    tacheB(void)
{
    int i=0;
    long j;
    puts("-----> DEBUT tache B");
    while (1) {
        for (j=0; j<30000L; j++);
        printf("=====> Dans tache B %d\n",i);
        i++;
    }
}

TACHE    tacheC(void)
{
    int i=0;
    long j;
    puts("-----> DEBUT tache C");
    while (1) {
        for (j=0; j<60000L; j++);
        printf("=====> Dans tache C %d\n",i);
        i++;
    }
}
```

```

TACHE    tacheD(void)
{
    int i=0;
    long j;
    puts("-----> DEBUT tache D");
    while (1) {
        for (j=0; j<120000L; j++);
        printf("=====> Dans tache D %d\n",i++);
        if (i==50) noyau_exit();
    }
}

int main()
{
    serial_init(115200);
    puts("Test noyau");
    puts("Noyau preemptif");
    // Initialisation du noyau, création et lancement de la tâche de départ
    start(tacheA);
    return(0);
}

```

Le résultat obtenu est le suivant. On voit que les tâches sont correctement préemptées et sont activées approximativement le même nombre de fois au cours de l'essai. L'attente active de chaque tâche n'importe donc évidemment pas sur le fait de se faire préempter ou non.

Test noyau

Noyau preemptif

```
-----> EXEC tache A
-----> DEBUT tache B
-----> DEBUT tache C
=====> Dans tache B 0
-----> DEBUT tache D
=====> Dans tache B 1
=====> Dans tache C 0
=====> Dans tache B 2
=====> Dans tache C 1
=====> Dans tache B 3
=====> Dans tache D 0
=====> Dans tache B 4
=====> Dans tache B 5
=====> Dans tache C 2
=====> Dans tache B 6
=====> Dans tache C 3
=====> Dans tache B 7
=====> Dans tache D 1
...
=====> Dans tache C 97
=====> Dans tache B 195
=====> Dans tache B 196
=====> Dans tache C 98
=====> Dans tache B 197
=====> Dans tache B 198
=====> Dans tache D 49
```

Sortie du noyau

```
Activations tache 0 : 4
Activations tache 1 : 348
Activations tache 2 : 347
Activations tache 3 : 347
Activations tache 4 : 0
Activations tache 5 : 0
Activations tache 6 : 0
Activations tache 7 : 0
```