TP Noyau TR ARM

On ouvre le fichier noyau.c pour l'étudier.

Première partie, Réalisation d'un mini noyau temps réel ARM

1ère partie : Ordonnanceur de tâches

Etudions le fichier NOYAUFIL.C:

-

*file_init() : initialise la file. _queue contient une valeur de tâche impossible, F_VIDE, indiquant ainsi que la file est vide.
*ajoute(n) : ajoute la tâche n en fin de file, elle sera la dernière à être activée
-> on ajoute la nouvelle tâche après celle en exécution

*suivant() : retourne la tâche à activer, et met à jour _queue pour qu'elle pointe sur la suivante.
*retire(n) : retire la tâche n de la file sans en modifier l'ordre.

file_init():

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

ajoute(n):

```
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;
}
```

suivant():

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

retire(n):

```
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;
}
```

affic_queue():

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

affic_file():

```
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);
}
```

Ecrire programme de test, TESTFIL.C:

```
#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();
}
```

résultat:

```
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
```

Nous avons repris l'exemple et nous arrivons grâce à nos fonctions à le reproduire correctement.

2ème partie : gestion et commutation de tâches

Fonctions de NOYAU.C complété des fonctions suivantes:

start(adr_tache):

```

```

récupère le pointeur de pile.
Puis pour chaque tâche dans la tableau contexte, on initialise leur etat à NCREE car aucune tâche n'est créée de base.
Pourquoi la tâche courante est initialisée à zéro ? certains noyauTR 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 se retrouve en 0 des différents tableaux utilisés dans le code (_contexte, compteurs d'activations). De plus, quand on crée une tâche (voir la fonction) on part dans int static appelé tache à -1 puis tache est indenté de 1 et passe donc à 0, qui est bien la première valeur donnée lors du start avec la première tâche. Puis lors des créations des tâches suivantes on aura bien la variable static tache à jour qui sera à nouveau indentée (pour la seconde tâche, tache = 0 qui passe à 1 et est utilisée pour la création)

Ensuite notre file est initialisé avec la fonction créée précédemment.

cree(adr_tache):

```

```

active(tache):

```

```

fin_tache():

schedule():

quand il n'y a plus de tâche courante à exécuter (_tache_courante = F_VIDE) et on sort du noyau avec noyau_exit().

noyau_exit():

Notre programme de Test, NOYAUTES.C:

Résultat

```
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

code NOYAUTES.C:

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

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

/*
 ** Test du noyau preemptif. Lier noyautes.c avec noyau.c et noyaufil.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");
  start(tacheA);
  return(0);
}
```