TP MI11 - Réalisation d'un mini noyau temps réel (Parties 5 et 6)

5ème partie : attente passive

Dans les tests précédents du mini noyau ARM, on construisait des délais actifs à l'aide de boucles if. On veut maintenant que la tâche s'endorme un certain délai avant de reprendre son travail (dans le cas du test, il s'agit d'écrire son numéro à l'écran) afin de libérer le processeur pour une autre tâche. Toutes les tâches « de travail » pouvant être endormies, il faut une tâche de fond qui prendra le relai quand cela sera nécessaire. Cela sera dévolu à la première tâche lancée qui ne devra donc plus se tuer quand elle aura fini de lancer les autres tâches (voir `tachedefond` dans le fichier `noyau_test.c`).

Fourniture

Un projet Eclipse est fourni comme point de départ.

Comme ajout par rapport au TP précédent, vous trouverez :

- une nouvelle variable `cmpt` dans la structure de donnée `CONTEXTE` qui permet de stocker la valeur courante décomptage permettant de savoir quand réveiller la tâche après une demande de délai ;
- deux fichiers `delay.h` et `delay.c` pour la définition des nouvelles fonctionnalités ;
- une nouvelle définition du nombre maximum de tâches `MAX_TACHES_NOYAU` dans `noyau_file.h`. la correction du nom a été appliqué partout ou nécessaire (préparation de la partie 6)
- l'appel à chaque tick d'horloge de la fonction `delay_process()` dans la fonction `scheduler()` (elle est vide dans votre projet).

A faire :

- exclure de la compilation `noyau_file_prio.c` et `noyau_test_prio.c` ;
- compléter les deux fonctions de `delay.c` en fonction des indications indiquées dans les commentaires ;
- penser à initialiser la variable `cmpt` du contexte de la tâche en cours de création.

Question de réflexion pour les plus rapides (revenir sur le sujet s'il vous reste du temps après la partie 6) **:** serait-il possible d'améliorer cette fonctionnalité en ne parcourant pas l'ensemble des tâches dans `scheduler()` pour vérifier si il faut traiter un délai ? Comment ?

6ème partie : files avec priorités

On veut compléter la méthode d'ordonnancement en introduisant une notion de priorité en plus du « tourniquet » déjà implémenté (file circulaire). Pour chaque niveau de priorité ***i***, l'objectif est de gérer un tourniquet identique à celui déjà présent. On veut **8** niveaux de priorité numérotés de **0** à **7**. On veut au maximum 8 tâches dans chaque niveau de priorité. La priorité **0** est la plus grande, **7** la plus petite.
Comme pour d'autres éléments du noyau, il faut des structures de données de gestion pour mettre en œuvre ces priorités. Le numéro de la tâche est contenu dans un mot `uint16_t` comme précédemment. Ce numéro encodera à la fois la priorité de la tâche et son identité dans le tourniquet de priorité ***i***.
On stocke dans un octet d'occupation des priorités `_file_u` le fait qu'au moins une tâche est présente dans le tourniquet de priorité ***i***. Si le bit ***i*** de cet octet est positionné, alors il existe au moins une tâche dans le tourniquet de priorité ***i***.
La tâche de démarrage du noyau aura toujours le numéro **63**. C'est donc une des huit tâches les moins prioritaires au démarrage du noyau. On va la considérer comme la tâche de fond. Attention ce choix implique que toutes les autres tâches doivent faire une pause lors de leur démarrage pour laisser la possibilité à la tâche de fond de lancer toutes les tâches.

Fourniture

Comme ajout par rapport à la partie précédente du TP, vous trouverez :

- un nouveau fichier à compléter `noyau_file_prio.c` dont les prototypes de fonctions sont les mêmes que celles de `noyau_file.c` ;
- un nouveau fichier de test `noyau_test_prio.c` ;
- dans le fichier `noyau_file_prio.c` :
  - une nouvelle définition des files comme un tableau de files : `_file[MAX_FILE][MAX_TACHES_FILE]`
  - une nouvelle définition du pointeur de queue sous la forme d'un tableau `_queue[MAX_FILE]` ;
  - une nouvelle variable `_file_u` qui sert à mémoriser les files ayant des tâches en exécution ;
  - un tableau `tab_num_file[256]` qui permet de déterminer sans calcul quelle est la file contenant une des tâches les plus prioritaires (si la variable `_file_u` est bien maintenue à jour).

A faire :

- les fonctions de gestion de files circulaires d'ordonnancement sont impactées par l'apparition des priorités. Les algorithmes sont les mêmes pour une file donnée mais il faut l'appliquer à la bonne file dont le numéro est encodé dans l'identité de tâche (dans les bits 3 à 5) ; On conseille l'utilisation de pointeur qui simplifieront les expressions d'accès au tableau des files circulaires ; penser à mettre à jour la variable `_file_u` quand vous ajoutez ou retirez une tâche d'une file ; complétez les fonctions du fichier `noyau_file_prio.c` ; pensez à modifier le nombre de files `MAX_FILE` dans `noyau_file.h` ;
- ajouter une nouvelle variable `uint8_t flag_tick` dans la structure `CONTEXTE` et trois nouvelles fonctions dans `noyau.c` :
  - `uint8_t tache_get_flag_tick(uint16_t id_tache)`
  - `void tache_set_flag_tick(uint16_t id_tache)`
  - `void tache_reset_flag_tick(uint16_t id_tache)`

permettant l'accès à ce flag depuis les autres parties du code (utilisées dans le code de test dans les fonctions de tâches pour ne générer qu'une seule écriture de la tâche par tick horloge) ;

- compléter le scheduler en faisant un set de ce flag, pour toutes les tâches (en exécution ?), à chaque tick d'horloge ;
- ajouter trois nouvelles variables dans la structure `CONTEXTE` :
  - `void *contexte_add` qui va permettre d'adjoindre des paramètres personnalisés à la tâche (utilisée dans le test pour passer deux paramètres à chaque tâche) ;
  - `uint16_t prio` qui va permettre d'enregistrer la priorité de la tâche ;
  - `uint16_t id` qui va permettre d'enregistrer l'identité de la tâche ;
- modifier la primitive `cree()` car on va changer le paradigme de choix de numérotation des tâches. En effet, on va maintenant laisser l'utilisateur choisir cette identité.
  - ajouter un premier paramètre `uint16_t id` à la primitive `cree()` ;
  - ajouter un deuxième paramètre `void* add` à la primitive `cree()` ;
  - modifier lignes nécessaires pour tenir compte du changement de paradigme et du pointeur sur les paramètres de personnalisation ;
- modifier la primitive `start()` qui lance forcément la première tâche avec l'identité 63. Attention au numéro de la première tâche courante.

Questions à se poser pour réaliser le TP :

⇨ Sachant une identité de tâche, comment obtenir le niveau de priorité de la tâche ?
⇨ Sachant une identité de tâche, comment obtenir le numéro de cette tâche dans sa file de priorité ?
⇨ Sachant l'identité de la tâche et donc sa priorité ***i*** (voir ci-dessus), comment mettre à jour l'octet `_file_u` lors des différentes modifications d'état d'une tâche.

Question de réflexion pour les plus rapide : faire une fonction permettant de changer la priorité (attention cela change aussi son identité et son numéro sur la file d'arrivée car son ancienne place est peut-être occupée.
Ce changement d'identité peut-il être problématique dans le cadre global du fonctionnement du micro-noyau ?