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

Introduction

Ce TP consiste à réaliser, sur une carte ARM ARMADEUS et en langage C un mini noyau temps réel très simplifié. Le but de ce noyau est de pouvoir répondre correctement au programme test proposé dans le fichier **noyau_test.c**.

Les services proposés à l'utilisateur dans un premier temps sont les suivants :
- commencer une première tâche : *start(tâche)*
- créer une tâche : *cree(tâche)*
- activer une tâche : *active(tâche)*
- appel au scheduler : *scheduler()*

Ces services sont ceux qui apparaissent dans le programme test ou qui seront utilisés par la suite. Dans un deuxième temps, on ajoutera des services de synchronisation et de communication inter-tâches.

Le noyau proposé est préemptif, c'est-à-dire que les appels au noyau peuvent être explicites dans les tâches ou les primitives du noyau, ou provoqués par le noyau temps réel. Il peut donc y avoir des appels au service d'interruption dus au système extérieur ou à l'horloge interne.

A chaque tâche est associé un contexte. Tous les contextes sont rangés dans un tableau. On ne peut avoir que 8 tâches au maximum.

Le contexte associé à chaque tâche est constitué de (structure **CONTEXTE** dans **noyau.h**) :
- une adresse de début de tâche,
- l'état de la tâche qui peut être :
  - non crée *NCREE*
  - créé ou dormant *CREE*
  - prêt *PRET*
  - en exécution *EXEC*
- un pointeur de pile initial,
- un pointeur de pile courant en mode *IRQ*.

Une tâche est décrite comme un sous-programme (fonction C) de type *TACHE*.

Tous les types, constantes et prototypes de fonctions utilisés dans le noyau sont définis dans les fichiers **noyau.h** et **noyau_file.h** ; il est donc **fortement** recommandé de consulter de façon approfondie ce fichier.

Les fichiers principaux à modifier sont :
- **noyau.c** noyau et fonctions associées
- **noyau_file.c** procédures de gestion de la file circulaire de tâches
- **qemu_versatile.x** fichier *elf* à compléter avec l'adresse de départ de la pile system
- **crt0.S** fichier de démarrage en assembleur, à compléter avec le nom de la fonction à brancher sur le vecteur d'interruption *irq*.

Vous devez avoir les fichiers suivants écrits par vos soins lors du TP précédent (les noms peuvent être différents mais, dans ce cas-là, attention aux INCLUDE dans les fichiers fournis :

- **versatile_interrupt.c** et **versatile_interrupt.h** doivent contenir du code permettant d'accéder aux interruptions ;
- **versatile_serial.c** et **versatile_serial.h** doivent contenir du code permettant d'accéder au port série ;
- **versatile_timer.c** et **versatile_timer.h** doivent contenir du code permettant d'accéder aux timers.

Les fichiers **serialio.*** fournis permettent des entrées et sorties sur le port série de la carte avec des fonctions de haut niveau comme *printf()*. On peut visualiser ces sorties par l'intermédiaire d'un terminal série adéquate (ex : Putty, telnet ou Cutecom).

1ère partie : Ordonnanceur de tâches

Son rôle est de gérer l'activité des tâches prêtes ou en exécution. Toutes les tâches ont la même priorité. La gestion de leur activité se fait en FIFO, c'est-à-dire que la plus ancienne dans la file sera la première activée.

La solution choisie dans ce noyau minimum consiste à mémoriser les numéros des tâches dans un tableau *_file* de dimension *MAX_TACHE*. L'indice d'un élément du tableau correspond à un numéro de tâche et l'élément du tableau à la tâche suivante. La variable *_queue* contient l'indice de la tâche active, de sorte qu'on puisse connaître la prochaine tâche à activer.
Dès qu'une tâche devient active, elle est automatiquement remise en fin de file, puisqu'elle sera la dernière à être réactivée.

Les fonctions qui gèrent cette file sont déclarées dans le fichier **noyau_file.h** (prenez bien soin de regarder les définitions qui y sont) et doivent être définies dans le fichier **noyau_file.c** :

- *file_init()* : initialise la file. *_queue* contient une valeur de tâche impossible, *F_VIDE*, indiquant ainsi que la file est vide.
- *file_ajoute(n)* : ajoute la tâche n en fin de file, elle sera la dernière à être activée
- *file_suivant()* : retourne la tâche à activer, et met à jour *_queue* pour qu'elle pointe sur la suivante.
- *file_retire(n)* : retire la tâche *n* de la file sans en modifier l'ordre.

Exemple

On suppose la situation initiale suivante :

Début				Fin
3	5	1	0	2

Tâche active : 2

Le tableau est donc :

			_queue					
Indice	0	1	2	3	4	5	6	7
_file	2	0	3	5		1		

suivant()

Début				Fin
5	1	0	2	3

Tâche active : 3

Le tableau est donc :

				_queue			
Indice	0	1	2	3	4	5	6	7
_file	2	0	3	5		1		

retire(0)

Début			Fin
5	1	2	3

Tâche active : 3

Le tableau est donc :

				_queue			
Indice	0	1	2	3	4	5	6	7
_file		2	3	5		1		

ajoute(6)

Début				Fin
5	1	2	3	6

Tâche active : 3

Le tableau est donc :

							_queue	
Indice	0	1	2	3	4	5	6	7
_file		2	3	6		1	5	

Le test s'imagine avant le code

Le principe de base d'un noyau, temps-réel ou non, est d'ordonnancer des tâches, stockées dans une structure de données. Si les fonctions de gestion de cette structure de données n'ont pas exactement le comportement attendu, c'est l'ensemble du noyau qui ne fonctionnera pas. Et ces éventuels problèmes peuvent survenir bien plus tard, dans des cas limites non pensés initialement.

Afin de s'assurer du bon fonctionnement des fonctions de gestion de la file, on se propose de réfléchir à des tests unitaires pour tester l'ensemble des fonctions. Ces tests sont à développer dans le fichier **noyau_file_test.c**.

Question 1.a : après avoir décrit brièvement ce que sont des tests et quelle est la spécificité des tests unitaires (5 lignes maximum), vous discuterez de la pertinence de réfléchir à la façon de tester un système avant de l'implémenter (5 lignes minimum).

Question 1.b : en fonction des explication données ci-dessus, proposez autant de cas de test unitaires pertinents pour chaque fonction de gestion de la file.

Algorithme

Avant de commencer l'implémentation des fonctions de gestion de la file dans le fichier **noyau_file.c**, il est important de réfléchir aux algorithmes de gestion de la file.

Question 1.c : décrivez en langage naturel l'algorithme des fonctions *ajoute(n), suivant()* et *retire(n)*. Il est vivement conseillé de réfléchir aux cas limites, correspondants normalement aux cas de tests de la section précédente.

Implémentation

Récupérez l'archive **qemu_versatile_noyau.tgz** sur Moodle, l'extraire. Cette archive contient les fichiers nécessaires pour cette première session du TP. Pour cette partie, vous pouvez travailler dans un projet linux classique mais je vous conseille de créer (ou réutilisez) un projet Qemu[1] comme pour le TP ARM précédent et récupérez les fichiers dans votre dossier projet.

Question 1.d : Maintenant que le système a été pensé en termes d'algorithme et de tests, commencez par implémenter vos cas de tests dans le fichier **noyau_file_test.c**.

Question 1.e : dans *file_init()*, à quelle valeur doit-on initialiser la variable *_queue* ?

Question 1.f : Une fois que vos cas de tests ont été développés, développez les fonctions définies dans le fichier **noyau_file.c**.

Question 1.g : Compilez le programme et testez. Tant que l'ensemble de vos cas de tests ne fonctionnent pas, il est inutile de passer à la deuxième partie.
Faire des captures des résultats et les expliquer dans le rapport.

[1] Avec de l'émulation, ce n'est pas beaucoup plus consommateur de temps. C'est différents sur une carte réelle ou chaque exécution prend du temps à cause du chargement sur la carte.

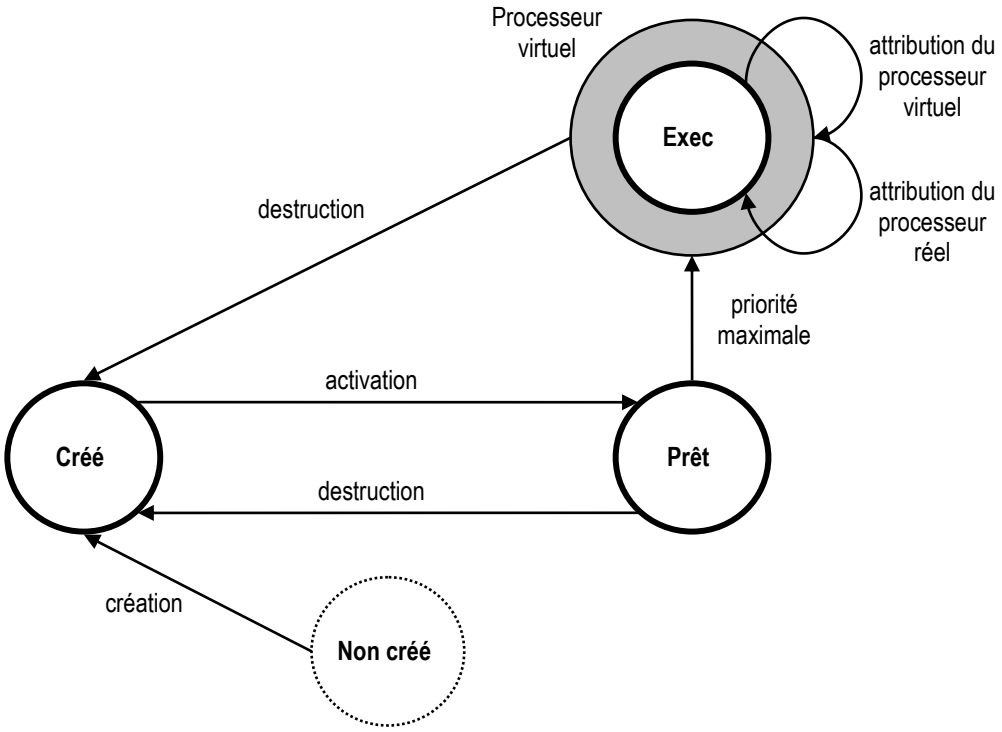
2ème partie : gestion et commutation de tâches

Cette partie du TP consiste à réaliser les primitives de gestion des tâches du mini noyau temps réel, ainsi que le système de commutation de tâches.

Etat des tâches

Les tâches peuvent prendre l'un des quatre états suivants :

- **Non créée (NCREE) :** la tâche n'existe pas, elle n'est pas connue du noyau.
- **Créée (CREE) :** la tâche est connue du noyau, une pile lui a été allouée ainsi qu'un identifiant, elle est prête à être activée. Elle reste dans cet état tant qu'un événement ne la fait pas évoluer.
- **Prête (PRET) :** une tâche à l'état prêt est candidate à l'exécution. Son lancement ne dépend que de sa priorité par rapport aux tâches en cours d'exécution ou aux autres tâches à l'état prêt. C'est l'ordonnanceur (scheduler) qui décide.
- **En exécution (EXEC) :** une tache en exécution est une tâche en possession du processeur virtuel ou du processeur réel. Dans un système monoprocesseur, plusieurs tâches de même priorité peuvent être en exécution mais une seule tâche est en possession à un instant donné du processeur réel. C'est l'ordonnanceur qui décide laquelle.



Description du contexte d'une tâche

Chaque tâche est associée à un contexte. Tous les contextes sont rangés dans le tableau *_contexte*. On ne peut avoir que 8 tâches au maximum.
Le contexte associé à la tâche est constitué de :

- Une adresse de début de la tache *tache_adr*
- L'état de la tache *status*
- Un pointeur de pile initial *sp_ini*
- Un pointeur de pile courant en mode *IRQ* : sp_irq

L'état du processeur (registres, pointeur d'instruction) doit être sauvegardé dans la pile de la tâche courante lors de l'appel au scheduler, que cet appel soit dû à l'horloge interne ou à l'application. La sauvegarde et la restitution du contexte dans le scheduler se fait donc en manipulant le pointeur de pile courant du mode d'interruption du processeur.

Services proposés

- démarrer le noyau et commencer une première tâche : *start(adr_tache)*
  - initialise les structures de données du noyau, met en place le gestionnaire d'interruption *scheduler*
  - crée et active la première tâche, dont l'adresse est passée en paramètres
- créer une nouvelle tâche : *cree(adr_tache)*
  - alloue un espace dans la pile à la tâche et lui attribue un identifiant, qui est retourné à l'appelant.
- activer une tâche : *active(tache)*
  - place la tache dans la file des tâches prêtes
- détruire une tâche : *fin_tache()*
  - change l'état d'une tâche active en CREE. La tâche peut alors être relancée à partir du début. Elle est retirée de la file des tâches prêtes.
- appel au gestionnaire de tâches (scheduler) : *schedule()*
- sortie du noyau : *noyau_exit()*

Chaque tâche est une fonction C, de type *TACHE*. Dans les primitives décrites ci-dessus, *adr_tache* représente l'adresse de la fonction associée à une tâche, et *tache* le numéro de la tâche.

Variables globales du noyau

- *_tos (top of stack)* : point de la pile à partir duquel il est possible d'allouer de l'espace pour une nouvelle tâche
- *_tache_c* : identifiant de la tâche courante, en exécution sur le processeur physique.
- _contexte[MAX_TACHES] : tableau de contextes des tâches
- compteurs[MAX_TACHES] : tableau permettant de comptabiliser individuellement le nombre d'activation par tâche.
- _ack_timer : drapeau permettant de différencier une activation du service d'interruption par le Timer ou le logiciel.

Implémentation

Les tests se font en cross-compilation sur la carte émulée par Qemu. En effet, le noyau ne peut pas s'exécuter sur Linux, il est développé pour des systèmes ARM simple.

Questions

Objectif : compléter le fichier **noyau.c** et tester le noyau au moyen du programme **noyau_test.c**.

Prélude

Question 2.a : définir la variable __sys_tos dans **qemu_versatile.x**. Elle correspond à l'adresse de début de la pile système et doit se situer à la fin de la première partie de la mémoire. Quelle valeur allez-vous choisir ?

Question 2.b : définir la fonction branchée sur le vecteur d'interruption @irq dans **crt0.S**.

Fonction *noyau_exit()*

Question 2.3 : quelle macro devez-vous utiliser pour désactiver les interruptions ? (Regardez dans **versatile_interrupt.h**). Expliquer son fonctionnement.

Question 2.4 : vous allez sortir du noyau, quelle instruction proposez-vous ici ? d'autres solutions sont possibles pour éviter une catastrophe. Lesquelles ?

Fonction *fin_tache()*

Question 2.5 : même question que Q2c.

Question 2.6 : trois lignes de code. Quelle est le statut d'une tâche que l'on tue (voir explications ci-dessus) ?
si on retire la tâche de la liste des tâches ordonnançables, d'où faut-il la retirer ?
Que se passe-t-il la fonction retourne ici ? que faut-il ajouter comme instruction pour éviter ce désagrément ?

Fonction *start()*

Question 2.7 : dans la boucle, à quelle valeur doit-on initialiser le statut des tâches au démarrage du noyau ?

Question 2.8 : quel est le rôle de la variable *_tache_c* ? A quelle valeur doit-on l'initialiser ? (pour répondre se reporter à la fonction *cree()* pour connaitre le numéro d'identité donné à la toute première tâche créée).

Question 2.9 : dans quel mode est votre processeur lorsqu'on arrive à cette ligne ? quel élément contient la valeur d'adresse du haut de la pile pour ce mode ? Initialiser la variable _tos avec cette valeur d'adresse.

Question 2.10 : utilisez la macro permettant de passer le processeur en mode IRQ.

Question 2.11 : le pointeur de pile n'a normalement pas été initialisé (sauf si vous avez essayé de le faire dans votre fonction **crt0.S**). Le faire ici si nécessaire.

Question 2.12: revenir en mode système.

Question 2.13 : utilisez la macro permettant d'interdire les interruptions. Pourquoi est-ce nécessaire en ce point ?

Question 2.14 : utilisez la fonction vous permettant d'initialiser le timer 0 à la période demandée.

Question 2.15 : utilisez la fonction vous permettant d'autoriser le timer 0 à déclencher des interruptions.

Fonction *cree()*

Question 2.16 : qu'est-ce qu'une variable *static* ? Quelle est la valeur réelle de la variable *tache* lors de son initialisation ? quelle est son rôle et pourquoi l'initialise-t-on à -1 ?

Question 2.17 : insérer la macro permettant de commencer une section critique (voir fichier ***.h**) ? quelle est la différence entre une section critique et un simple blocage des interruptions ?

Question 2.18 : insérer l'instruction qui permet d'incrémenter le numéro d'identité de tâche pour l'attribuer à la tâche en cours de création.

Question 2.19 : que se passerait-il si une tâche avait un numéro d'identité en dehors de la plage autorisée ? si cela se produit, on arrête le noyau (solution la plus simple mais pas la plus élégante et la plus sure).

Question 2.20 : on doit allouer un espace de pile à chaque nouvelle tâche. Dans le contexte de la tâche, quelle variable utiliser pour enregistrer l'adresse de début de cette pile ? Faire l'initialisation.

Question 2.21 : dans ce noyau, on a décidé de scinder la pile de chaque tâche en deux zones : la pile système et la pile IRQ. Quel est l'espace totale occupé par la pile d'une tâche ? (attention des définitions sont données dans le fichier .h).
Modifiez en conséquence la variable de mémorisation du sommet de la pile.

Question 2.22 : insérer la macro permettant de finir une section critique.

Question 2.23 : enregistrez l'adresse de la fonction représentant le code de la tâche dans le contexte associé à cette tâche.

Question 2.24 : passer le statut de la tâche à CREE dans le contexte associé à cette tâche.

Fonction *active()*

Question 2.25 : il n'est pas possible d'activer une tâche dont le statut serait NCREE. Dans ce cas on arrête le noyau (solution la plus simple mais pas la plus élégante et la plus sure).

Question 2.26 : insérer la macro permettant de commencer une section critique.

Question 2.27 : dans le cas où la tâche est dans l'état CREE, on peut l'activer. Dans le contexte de la tâche, modifier son statut.
Ajoutez la tâche à la file circulaire des tâches à ordonnancer.
Comment prendre en compte cette modification dans la liste des tâches à ordonnancer ? Appeler la fonction permettant cette action. Dans quel code reviendrait-on si on ne le faisait pas ? Trouvez-vous cela gênant dans le cadre de l'ordonnancement choisi pour ce noyau ; dans le cadre d'un ordonnancement par priorité ? En insérant la fonction demandée, dans quel code revient-on après la fin de la fonction en question ?

Question 2.28 : insérer la macro permettant de finir une section critique.

Fonction *schedule()*

Un gestionnaire d'interruption est logiquement fait pour être appelé lorsqu'une interruption matérielle se déclenche. Mais il peut être utile de l'appeler en dehors de ce contexte parce qu'on veut que le code que ce gestionnaire soit exécuté à un moment précis dans le code. Mais ce gestionnaire est fait pour être exécuter en mode IRQ et non en mode système. Il faut donc

préparer le processeur pour cet appel[2].
Il est aussi nécessaire dans le gestionnaire d'interruption de faire la différence entre un appel matériel ou logiciel. Pour cela, on utilise un flag (ici *_ack_timer*).

Question 2.29 : insérer la macro permettant de commencer une section critique.

Question 2.30 : si une interruption matérielle est perçu par la valeur 1 pour la variable *_ack_timer*, quelle valeur va-t-on naturellement lui donner ici ?

Question 2.31 : utilisez la macro permettant de passer le processeur en mode IRQ.

Question 2.32 : écrire les instructions assembleur nécessaires permettant de simuler un appel matériel du gestionnaire d'interruption (4 lignes au total).

Question 2.33 : utilisez la macro permettant de passer le processeur en mode système.

Question 2.34 : insérer la macro permettant de finir une section critique.

Fonction *scheduler()*

Cette fonction est le gestionnaire d'interruption du noyau. Sa fonction principale est de sauvegarder le contexte de la tâche interrompue à l'entrée dans le gestionnaire et de le restituer pour la tâche qui va être remise sur le processeur si celle-ci a déjà commencé son exécution sur une période précédente.

Question 2.35 : écrire les instructions assembleur nécessaires permettant de sauvegarder le contexte de la tâche interrompue sur sa pile (5 lignes). **Attention**, on veut sauvegarder le contexte du processeur en mode système dans la partie IRQ de la pile (donc pointé par le pointeur de pile IRQ en cours). Cela nécessite l'utilisation d'une construction un peu particulière permettant une commutation vers le contexte système à l'exécution de l'instruction (bien relire la documentation sur les exceptions). Expliquer vos choix et leurs effets.

Question 2.36 : que faut-il penser à faire sur le timer si l'interruption est matérielle. Sinon (interruption émulée par le logiciel) que faut-il ne pas oublier de faire sur le flag si on veut pouvoir exécuter la prochaine interruption matérielle correctement.

Question 2.37 : sauvegarder le registre pointeur de pile (IRQ) dans le contexte de la tâche.

Question 2.38 : faire évoluer la variable *_tache_c* pour la positionner à la valeur d'identité de la prochaine tâche en exécution.

Question 2.39 : si la valeur de *_tache_c* indique que la prochaine tâche n'existe pas (en général, il n'y a plus de tâche à ordonnancer ou il y a un bug), arrêtez le noyau (solution la plus simple mais pas la plus élégante et la plus sure).

Question 2.40 : initialisez le pointeur de contexte sur le contexte de la tâche sélectionnée pour être mise sur le processeur.

[2] Lorsque qu'il y a une interruption matérielle, on rappelle que la commutation en mode IRQ est automatique ; c'est un comportement naturel du matériel ARM.

Question 2.41 : quelle est la particularité de la pile IRQ d'une tâche dont le statut est PRET par rapport à une tâche est EXEC ?
Pour chaque ligne de commentaire, écrire la commande nécessaire.
Par rapport aux commentaires, quelle est l'adresse de début donnée à la partie IRQ de la pile de la tâche. En déduire la valeur d'adresse que vous allez attribuer au pointeur de pile de la partie système de la pile de la tâche après le passage du processeur en mode système.
Quelle instruction allez-vous utiliser pour lancer le code de la tâche ? (attention on passe un pointeur sur la fonction). Est-ce que vous pensez que le code va revenir à cette section du code (instruction après l'instruction de lancement de la tâche) à un moment donné de son exécution ?
N'oubliez pas l'instruction dans le *else* qui permet de donner une nouvelle valeur au pointeur de pile IRQ correspondant au pointeur de pile sauvegardé dans le contexte de la tâche lors de sa précédente préemption.

Question 2.42 : écrire les instructions assembleur nécessaires permettant de restaurer le contexte de la tâche à mettre sur le processeur et relancer la tâche (6 lignes). Encore une fois, attention à la syntaxe.
Expliquer vos choix et leurs effets.

Question 2.43 : La commutation de contexte se fait par commutation de pointeur de pile IRQ. Comment se fait la commutation du pointeur de pile système des tâches (dans laquelle on retrouve les variables locales de la tâche ainsi que les éléments nécessaires aux appels de fonctions) à chaque commutation de tâche ?