1. Introduction

Le but du labo OS est d'explorer les fondements d'un système d'exploitation multitâche simple pour ARM en le réalisant soi-même (do it yourself :-)

La cible matérielle est une carte LPC55S69-EVK. L'ensemble du code s'exécutera sur le Corex-M33 Core 0 du microcontrôleur LPC55S69 de NXP.

Le logiciel fait apparaître une organisation en deux parties :

- Le **code applicatif** qui s'exécute en mode **Thread unprivileged**.
  - Plusieurs **tâches** (*processus*) peuvent s'exécuter en pseudo-parallèle.
  - L'accès de la couche applicative aux services du système d'exploitation (création de tâche, de sémaphore, …) se fait via les **appels systèmes** (SVC = SuperVisor Call).
  - Les interruptions sont autorisées.
- Le **système d'exploitation** composé :
  - du **noyau** qui s'exécute en mode **Handler** (privileged)
    - Il offre derrière les appels systèmes une gestion des tâches et des sémaphores, ainsi que le moyen de démarrer la commutation des tâches. Les SVC sont non interruptibles par les IRQ des périphériques.
    - Il gère la **commutation de tâches** sur interruption du timer système en mode tourniquet (round-robin) ou sur des évènements liés à l'utilisation des sémaphores ou de la fonctionnalité de temporisation. Il décide de la tâche applicative qui peut utiliser le processeur à un instant donné.
  - d'une **couche driver** qui s'exécute en mode **Thread unprivileged** qui permet de la communication entre la partie applicative et le matériel à travers une interface standard. On s'appuie sur les outils de synchronisation fournis par le noyau (sémaphores) pour offrir un fonctionnement adéquat, que le périphérique fonctionne sur interruption ou pas.

Lors de l'appel de la fonction `main`, le processeur se trouve en mode **Thread privilégiés** avec **msp** comme pointeur de pile. Les IRQ des périphériques sont inhibées. Seules les SVC sont autorisées. Une initialisation minimale du système a été effectuée.

Le projet est géré (génération du makefile, du script de l'éditeur de lien, des fichiers à compiler) par l'IDE MCUXpresso de NXP (un clône d'Eclipse). Le répertoire du projet est organisé de la manière suivante (je n'insiste que sur les fichiers importants pour le projet) :

```
/répertoire racine du projet/
 +- board/                          : config du microcontrôleur effectuée avant l'entrée
 |                                    dans le «main»
 +- CMSIS/                          : couche d'abstraction du processeur [source : ARM]
 +- device/
 |  +- LPC55S69_cm33_core0.h        : définition des IRQn, structures des périphériques
 |  +- target.[ch]                  : drivers au niveau OS
 |  +- vfs.[ch]                     : interface générique de drivers OS
 +- drivers/                        : drivers de bas niveau du microcontrôleur
 +- kernel/
 |  +- kernel.[ch]                  : code du noyau
 |  +- list.[ch]                    : implémentation de listes circulaires
 +- startup/
 |  +- startup_lpc55s69_cm33_core0.c : code de startup, table des vecteurs
 |  +- svc.s                        : SVC_Handler et PendSV_Handler
 |
 |                               C O D E   S Y S T E M E
 | ------------------------------------------------------------------------------
 |                           C O D E   U T I L I S A T E U R
 |
 +- source/
    +- oslib.[ch]                    : bibliothèque d'appels systèmes
    +- main.c                        : code applicatif
```

Le système sera construit de manière progressive : le fichier `source/main.c` est subdivisé en plusieurs parties encapsulés dans les macros **#ifdef** … **#endif**. Chaque partie contient un exemple de code applicatif utilisé pour tester la mise en œuvre d'une fonctionnalité de l'OS en cours de développement. On choisit le programme de test en définissant avec un **#define**, en haut du fichier `main.c`, le label correspondant à la partie qu'on souhaite inclure dans la compilation.

Les différents labels sont :

`MAIN_TEST`	:	test de la mécanique d'un appel système
`MAIN_EX1`	:	test de la mise en route de la première tâche et de la commutation de tâche
`MAIN_EX2`	:	idem, mais toutes les tâches exécutent le même code
`MAIN_EX3`	:	test d'un sémaphore
`MAIN_EX4`	:	exemple d'utilisation d'un sémaphore comme mutex
`MAIN_EX5`	:	test de la fonctionnalité de temporisation
`MAIN_EX6`	:	idem, mais 2 tâches temporisées
`MAIN_EX7`	:	test de la terminaison d'une tâche
`MAIN_EX8`	:	test de l'interface virtuelle de fichiers
`MAIN_EX9`	:	la led RGB comme périphérique
`MAIN_EX10`	:	le bouton USR comme périphérique avec la gestion de l'appui sur interruption

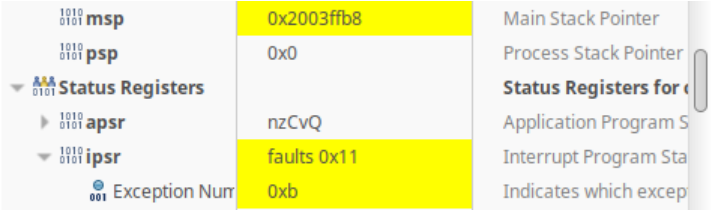
2. Le noyau

2.1. Les appels systèmes

MAIN_TEST

Il s'agit ici de tester la mécanique des appels système et de compléter le fichier `source/oslib.c`.

a. Exécuter en pas à pas le programme. Noter en particulier le changement de mode au moment de l'exécution de l'instruction **svc**. Visualiser et expliquer les modifications de valeur des registres `msp`, `lr` et `ipsr`.



Continuer en pas à pas jusqu'à l'exécution de la fonction `sys_add`. Vérifier que les paramètres passés sont les mêmes que ceux qui ont été passés à `test_add`. Terminer l'exécution en pas à pas. Vérifier qu'à la sortie du `SVC_Handler` on revient à la fonction `test_add` dans le mode de départ, et qu'on peut récupérer le résultat de l'addition.

b. Compléter les appels système dans le fichier `source/oslib.c`. Compléter la fonction `svc_dispatch()` dans le fichier `kernel/kernel.c` pour que les fonctions d'implémentattion côté noyau soient correctement appelés. Les noms des fonctions qui implémentent un appel système côté noyau sont définis par `sys_`*appel_systeme*. Par exemple, l'implémentation de l'appel système `os_start` est la fonction `sys_os_start`. *On ne demande pas de réaliser l'implémentation des appels système (pour le moment …).*

Les appels systèmes considérés sont :

appel système	:	numéro	Commentaire
os_alloc	:	1	implémentation noyau : `malloc`
os_free	:	2	implémentation noyau : `free`
os_start	:	3	
task_new	:	4	
task_id	:	5	
task_wait	:	6	
task_kill	:	7	
sem_new	:	8	
sem_p	:	9	
sem_v	:	10	

c. Tester la mécanique d'appel pour quelques appels systèmes (pour vérifier qu'on arrive bien jusqu'aux fonctions d'implémentation et que les paramètres sont correctement passés).

2.2. Création et lancement d'une tâche

MAIN_EX1

Une **tâche** est une entité de code qui peut s'exécuter indépendamment du reste du programme. Elle est caractérisée par le code qui est exécuté (la fonction passée en paramètre à l'appel système `task_new`) et un contexte processeur (valeur des registres).

Pour garder une trace des informations concernant une tâche, on lui associe, côté noyau, un *descripteur de tâche* (*Task Control Block* ou *TCB*) défini dans le fichier `kernel/kernel.h` par :

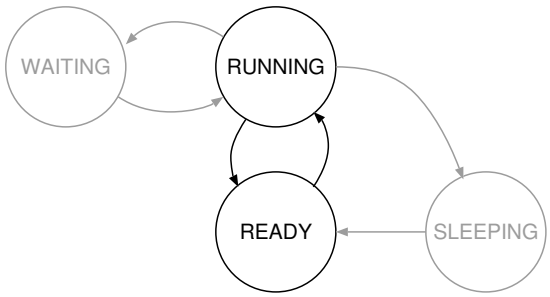
```
typedef struct _Task {
    struct _Task*   prev;       // prev task
    struct _Task*   next;       // next task
    uint32_t        id;         // identifier
    TaskState       status;     // task state : running, ready, ...
    uint32_t*       sp;         // current stack pointer
    uint32_t*       splim;      // stack limit
    int32           delay;      // wait delay
} Task;
```

Lorsque plusieurs tâches coexistent, les descripteurs de tâche sont (doublement) chaînés entre eux (pointeurs `prev` et `next`) pour former une liste circulaire (le dernier pointe sur le premier).

Les champs restants de la structure sont :

`id` identifiant associé au processus au moment de sa création à partir de la variable globale `id` définie dans `kernel/kernel.c`.

`status` état de la tâche. Elle peut être en train de s'exécuter (TASK_RUNNING), prête à s'executer mais ne disposant pas du processeur (TASK_READY), bloquée en attente de la fin d'une temporisation (TASK_SLEEPING), bloquée en attente devant un sémaphore (TASK_WAITING). Les constantes symboliques sont définies dans `kernel/kernel.h`. Les changements d'état des tâches se font suivant le graphe suivant (on précisera au fur et à mesure comment)



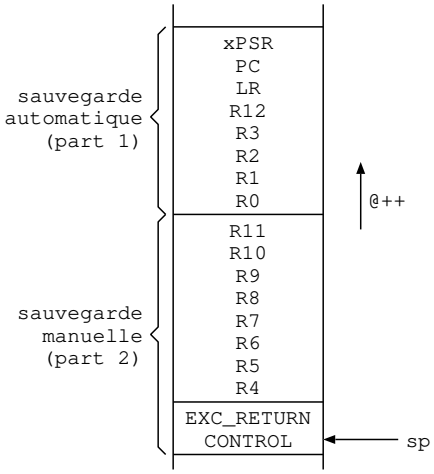
La variable globale `tsk_running` (définie dans `kernel/kernel.c`) pointe à chaque instant sur le descripteur de la tâche en train de s'exécuter.

Remarque : les tâches ne sont démarrées qu'après l'appel de `os_start()`.

`splim` adresse du bloc alloué dynamiquement et servant de pile pour le code de traitement de la tâche. Ce champ définit également la valeur limite que peut prendre le pointeur de pile.

`sp` lorsque la tâche n'est pas dans l'état *RUNNING*, `sp` représente la valeur du pointeur de pile associé à la tâche. Il pointe sur le contexte sauvegardé lors de la dernière interruption de la tâche.

Le contexte sur la pile est constitué de l'ensemble des valeurs des registres du processeur et a la structure suivante :

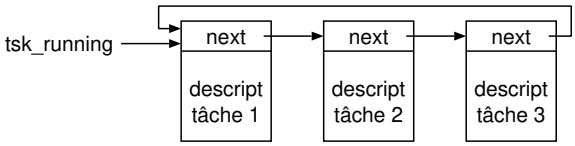


Chaque valeur empilée est codée sur 32 bits (4 octets). `sp[6]` représente la 7e valeur sur 32 bits en partant de la position de `sp`, donc permet d'accéder à la valeur `R8`. `EXC_NUMBER` représente le code de retour de l'exception qui permettra de revenir dans le code de traitement de la tâche. `CONTROL` représente la valeur qui sera copiée dans le registre de `CONTROL` du processeur avant de faire le retour au code de traitement de la tâche.

Lorsque la tâche s'exécute (*RUNNING*), la valeur `sp` n'est probablement plus en cohérence avec celle du pointeur de pile.

`delay` durée restante de la temporisation en cours pour cette tâche.

a. **Création de tâches** : l'implémentation noyau `sys_task_new` de l'appel système `task_new` permet d'allouer dynamiquement un descripteur pour la tâche créée, ainsi que la pile (allouer les deux en une seule fois), puis initialise les champs appropriés et insère le descripteur dans la **liste des tâches prêtes** dont le premier nœud est pointé par la variable globale `tsk_running`. En clair quand on a créé 3 tâches, on doit avoir :

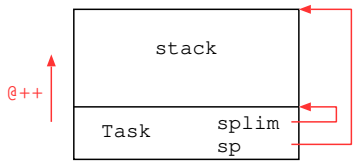


La fonction renvoie l'identifiant de tâche `id` ou -1 si la structure n'a pu être allouée.

On utilisera l'allocateur mémoire intégré à la libc associée au compilateur `malloc`[1]. Initialiser le champ `id` à partir de la variable globale `id` de telle sorte que chaque tâche ait un identifiant différent. Le champ `delay` pourra être initialisé à 0.

Le chaînage des blocs (pointeur `prev` et `next`) sera réalisé en utilisant les fonctions de manipulation de listes circulaires des fichiers `kernel/list.[ch]` (un exemple d'utilisation est fourni).

Il est nécessaire de faire pointer le champ `splim` en bas du bloc mémoire alloué pour la pile, puis le pointeur de pile en haut du bloc alloué pour la pile. L'empilement d'une valeur est réalisé en décrémentant de 4 octet le pointeur de pile, puis en écrivant la valeur à empiler (4 octets). Lorsque des valeurs sont empilées, le pointeur de pile descend donc dans les valeurs d'adresse. Il pointe sur la dernière valeur empilée. Le dépilement consiste à lire la valeur pointée par `sp` et à incrémenter `sp` de 4 octets.



[1] Il n'est pas possible d'appeler un appel système dans un appel système : cela provoque une UsageFault, qui sera promue en HardFault, si le support des UsageFault n'a pas été configuré

Les tâches sont créées avec l'état *READY* et il nécessaire de leur adjoindre un contexte minimal sur la pile qui permettra à la tâche de démarrer. Il faut positionner le pointeur de pile de manière à stocker ce contexte. Toutes les valeurs des registres n'ont pas besoin d'être initialisées. Il faut définir au minimum:

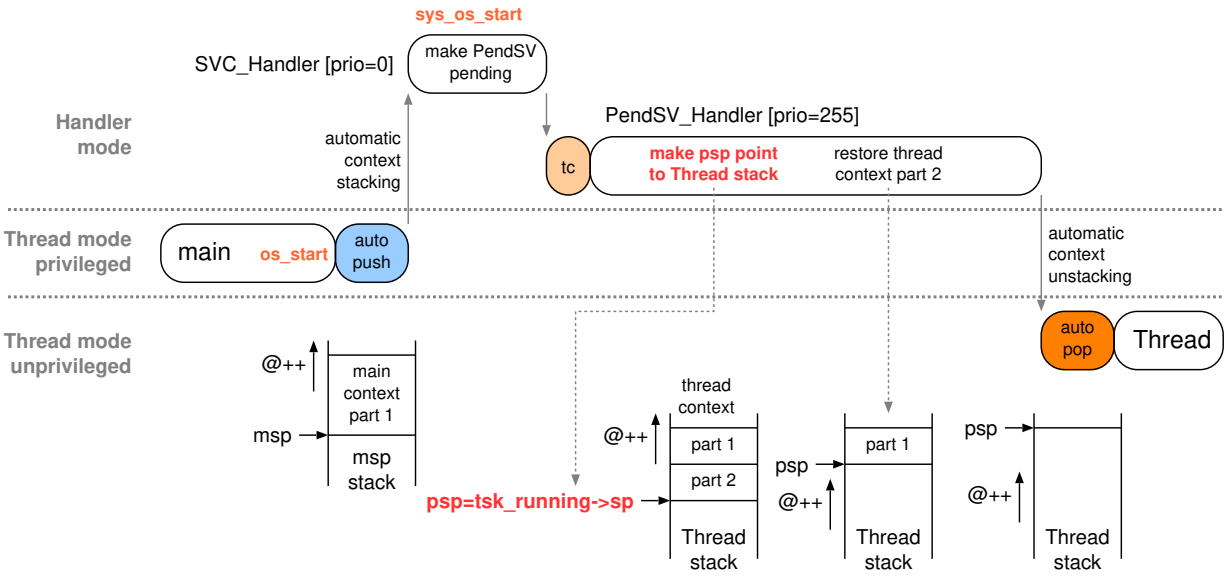
- `xPSR` : valeur du registre d'état au démarrage (juste le bit T),
- `PC` : qu'est-ce que la tâche doit exécuter ?
- `CONTROL` : la tâche s'exécute en mode **Thread unprivileged**
- `EXC_RETURN` : le code de retour de l'exception qui doit permettre le «retour» au code de traitement de la tâche, en considérant le fait que la tâche s'exécute en mode **Thread unprivileged** et utilise le pointeur de pile **psp**.

A ce stade, il devrait être possible, à l'issue des différents appels à `task_new` effectués dans la fonction `main` de vérifier la constitution de la liste de tâches pointée par la variable globale `tsk_running`.

b. Compléter l'implémentation noyau de l'appel système `task_id` qui renvoie l'identifiant de la la tâche pointée par `tsk_running`.

c. **Démarrage de la première tâche** : on change l'état en *RUNNING* et on réalise une modification du contexte processeur. Cette modification est confiée à l'exception PendSV qui a la priorité NVIC minimale. Ainsi ce sera la dernière exception traitée avant le retour au mode Thread. Pour provoquer la modification de contexte, il suffit de déclencher une exception de type Pend SV. C'est ce que fait la fonction `sys_switch_ctx()`.

Du point de vue transfert d'information, la routine de traitement de l'exception `PendSV_Handler` utilise le pointeur de pile `tsk_running->sp` pour réaliser le dépilement de la partie «manuelle» du contexte (part 2), ainsi que du registre `CONTROL` et du code de retour. Le pointeur `psp` du processeur est modifié pour pointer sur la pile de la tâche. Il est possible également sur le Cortex-m33 de modifier le registre `psplim` pour obtenir un garde-fou contre les débordements de pile[1].



Finalement, la partie «automatique» (part 1) sera automatiquement dépilée, et on devrait alors se retrouver dans le code de la tâche en mode Thread unprivileged avec `psp` comme pointeur de pile.

Compléter la fonction `sys_os_start` et vérifier qu'on est capable d'atteindre le début du code de la première tâche (mettre un point d'arrêt, peut être aussi dans PendSV_Handler pour vérifier qu'on passe bien par là aussi). Vérifier aussi que la première tâche s'exécute correctement (notamment l'appel système qui permet d'obtenir l'identifiant de la tâche). Une seule tâche a la possibilité de s'exécuter. On va faire mieux …

[1]Vou pouvez tester ce garde-fou en diminuant la taille de la pile de la tâche à 256 octets.

2.3. Commutation de tâches

La *commutation* entre deux tâches consiste à *interrompre la tâche en train de s'exécuter* puis *à donner le contrôle du processeur à une autre tâche*. Ces commutations peuvent avoir lieu

– sur les interruptions périodiques du Timer SysTick qui exécute la callback `sys_tick_cb` avec une période SYS_TICK configurée en ms (fichier `kernel/kernel.c`).

– au cours d'un appel système dans une tâche ou une routine d'interruption (voir sémaphore).

Dans les deux cas, la commutation doit indiquer quelle est la nouvelle tâche (`tsk_running`), garder une trace de la tâche qui vient de perdre la main `tsk_prev`, puis déclencher l'exception PendSV qui effectuera la commutation effective des contextes.

a. Analyser la routine de traitement de l'exception `PendSV` (`PendSV_Handler` dans le fichier `startup/svc.s`) et indiquer les actions réalisées au niveau de `tsk_prev` et `tsk_running`.

b. **Algorithme du tourniquet (Round Robin)** : compléter la callback `sys_tick_cb` (dans `kernel/kernel.c`) pour qu'à chaque appel, le processeur soit donné à la tâche suivante dans la liste. Tester le fonctionnement (Mettre des points d'arrêts, jouer éventuellement sur la valeur de SYS_TICK).

2.4. Réentrance

MAIN_EX2

- Expliquer à quoi fait référence la réentrance. Quelle condition(s) est(sont) nécessaire(s).
- Tester l'exemple. Expliquer pourquoi il se comporte de la même manière que le code précédent, bien que chaque tâche utilise la même fonction de traitement.

2.5. Les sémaphores

MAIN_EX3

Un **sémaphore** est un compteur de jetons associé à une liste de tâches bloquées en attente de jeton. Il est représenté par la structure suivante définie dans `kernel/kernel.h`

```
typedef struct _Semaphore {
        int32_t count;
        Task_t * waiting;
} Semaphore;
```

Le compteur `count` indique le nombre de jetons disponibles. Une valeur négative du compteur indique le nombre de tâche bloquées dans la liste `waiting` qui est gérée en FIFO.

Trois appels système sont implémentés pour manipuler les sémaphores :

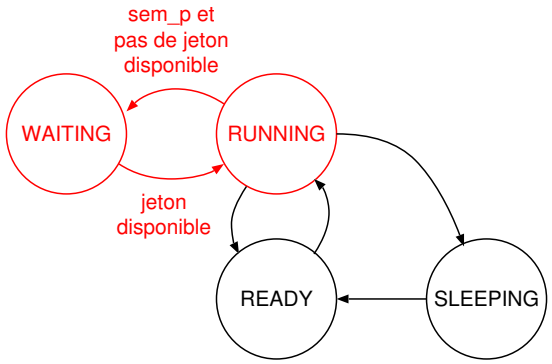
```
/* sem_new: create a semaphore
 *  init   : initial value
 */
Semaphore_t * sem_new(int32_t init);

/* sem_p: take a token */
void sem_p(Semaphore_t * sem);

/* sem_v: release a token */
void sem_v(Semaphore_t * sem);
```

L'implémentation est réalisée de la manière suivante

`sem_new`	:	création du sémaphore par allocation dynamique et initialisation du compteur et de la liste (aucun processus en attente).	
`sem_p`	:	décrémenter le compteur si compteur<0 (plus de jetons) alors bloquer la tâche qui a exécuté `sem_p` finsi	la tâche qui exécute l'appel système demande un jeton et devient bloquée s'il n'y en a plus de disponible.
`sem_v`	:	incrémenter le compteur s'il y a des tâches en attente alors débloquer la première tâche en attente et lui donner **immédiatement** le processeur. finsi	la tâche qui exécute l'appel système remet un jeton, ce qui peut éventuellement débloquer une tâche en attente d'un jeton.



Bloquer une tâche veut dire qu'on ne lui donne plus de temps processeur pour s'exécuter. Pour ce faire, il suffit de la changer de liste : la transférer de la liste `tsk_running` à la liste `waiting` du sémaphore et réaliser la commutation de contexte associée. Débloquer la tâche consiste en l'opération dans l'autre sens.

a. Compléter l'implémentation des appels systèmes `sem_new`, `sem_p` et `sem_v`. Tester le fonctionnement correct des sémaphores à l'aide du code d'exemple.

L'application met en place deux taches :

- la tâche 2 attend l'appui sur le bouton USER de la carte et met un jeton au sémaphore *sem* quand il y a appui,
- la tâche 1 attend devant le sémaphore *sem* un jeton correspondant à l'évènement «on a appuyé sur un bouton». Quand on reçoit cet évènement, la tâche est débloquée, la variable cpt incrémentée et affichée sur le terminal. Le fonctionnement de la tâche 1 est donc synchronisé par rapport à la tâche 2.

On fera particulièrement attention aux commutations provoquées par les appels systèmes `sem_p` et `sem_v` lors du débogage du code.

b. Pour assurer **l'exclusion mutuelle** à une ressource, il est possible d'utiliser le sémaphore en **mutex**[1].

MAIN_EX4

Tester l'exemple. Vérifier, en particulier, qu'à un instant donné, il n'y a qu'une seule tâche dans la **section critique**.

2.6. Gestion du temps

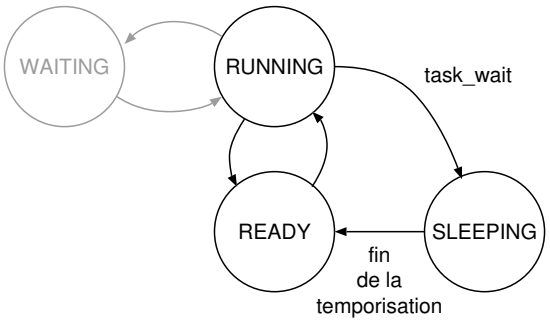
MAIN_EX5

Pour temporiser le fonctionnement d'une tâche, il est envisageable de la ralentir en lui faisant exécuter du

[1]De nombreux systèmes implémentent le mutex comme un sémaphore spécial ayant un compteur qui sature à 1, ce qui évite que davantage de Jetons puissent être introduits par erreur.

code qui ne sert à rien (une boucle de 10000 itérations vides par exemple). Cette solution présente toutefois l'inconvénient d'utiliser du temps CPU, même lorsque la tâche ne fait rien d'utile (autant le donner à des tâches qui en feront un usage intelligent). Par ailleurs, cette méthode ne permet pas de fixer de manière précise la durée de la temporisation.

On propose ici d'implémenter l'appel système `task_wait` qui permet *d'endormir une tâche* pendant une durée multiple de la période d'appel de la callback `sys_tick_cb` du timer SysTick (définie par la macro `SYS_TICK`=10 ms dans le fichier `kernel/kernel.c`).



Le descripteur de la tâche qui exécute l'appel `task_wait` est sorti de la liste des tâches prêtes et inséré dans la liste des tâches endormies pointée par la variable globale `tsk_sleeping` définie dans `kernel/kernel.c`. La valeur de la temporisation est codée dans le champ `delay` du descripteur de tâche.

A chaque interruption du timer, la liste des tâches endormies est parcourue, et la valeur `delay` est diminuée de `SYS_TICK`. Lorsque le champ `delay` $\leq 0 \Rightarrow$ la temporisation est terminée et la tâche peut être remise (en queue) dans la liste des tâches prêtes.

a. Implémenter l'appel système `task_wait` et vérifier son fonctionnement à l'aide du programme d'exemple.

b. MAIN_EX6

Vérifier le fonctionnement de votre code sur cet exemple un peu plus complexe qui permet à deux tâches d'utiliser la fonctionnalité de temporisation. Quelle est l'utilité de la tâche 3 ?

2.7. Terminaison d'une tâche

MAIN_EX7

Dans ce fichier, la tâche 4 doit logiquement se terminer à la sortie de la fonction `tache4()`, sauf que la sortie se passe mal … parce qu'on n'a pas prévu ce cas.

Compléter l'implémentation de l'appel système `task_kill` qui permet à une tâche de se terminer explicitement. Proposer et coder une solution permettant de faire en sorte que cette terminaison devienne implicite dans le cas où la fonction de traitement de la tâche se termine et de donner la main à une autre tâche.

3. Intégration des périphériques

3.1. Couche VFS

Le rôle de la couche VFS est d'offrir à l'utilisateur du système d'exploitation un accès au matériel de manière abstraite via une interface standardisée (`open`, `close`, `read`, `write`, `ioctl`, `lseek` et `readdir`) permettant de manipuler les notions de fichier et de répertoire sans se soucier des détails d'implémentation.

Chaque fichier ou répertoire est caractérisé par un chemin d'accès dans une arborescence de répertoires. La fonction `int open(char *path, int flags)` permet d'associer à un nom de fichier ou de répertoire un descripteur de fichier (`fd` : *file descriptor*) qui est fourni en paramètre de retour (-1 si erreur). Ce descripteur de fichier[1] est utilisé par toutes les autres fonctions de manipulation des fichiers ou répertoires :

[1] le terme *descripteur de fichier* pour l'entier renvoyé par `open` est un peu abusif. C'est plutôt un index qui permet d'accéder au descripteur de fichier de structure *FileObject* dans la table `opened_fds`.

```
/* open : returns a file descriptor for path name */
int open(char *path, int flags);

/* close : close the file descriptor */
int close(int fd);

/* read : read len bytes from fd into buf, returns actually read bytes */
int read(int fd, void *buf, size_t len);

/* write : write len bytes from buf to fd, returns actually written bytes  */
int write(int fd, void *buf, size_t len);

/* ioctl : set/get parameter for fd */
int ioctl(int fd, int op, void** data);

/* lseek : set the offset in fd */
int lseek(int fd, unsigned int offset);

/* readdir : iterates over directory fd, returns 0 when ok, -1 else */
int readdir(int fd, DIR **dir);
```

Côté VFS, pour chaque fichier ou répertoire ouvert, on alloue dynamiquement une structure `FileObject` (`oslib/vfs.h`) :

```
typedef struct _FileObject FileObject;

struct _FileObject {
        char *          name;           /* name of the file */
        unsigned int    flags;          /* file characteristics */
        unsigned int    offset;         /* offset in file */
#ifdef _FAT_H_
        FIL *           file;
        DIR *           dir;
#endif
        Device *        dev;            /* device driver */
};
```

Cette structure permet de caractériser un fichier ou un répertoire en conservant les informations :

- `name` : le nom du fichier,
- `flags` : un champ de bits (voir `source/oslib.h`) indiquant le type d'accès souhaité (O_READ ou O_WRITE pour nous) et le type d'objet dont il s'agit :
  - `F_IS_DIR` : il s'agit d'un répertoire
  - `F_IS_ROOTDIR` : c'est le répertoire racine "/"
  - `F_IS_DEVDIR` : c'est le répertoire spécial "/dev"
  - `F_IS_FILE` : il s'agit d'un fichier
  - `F_IS_DEV` : c'est un fichier de périphérique matériel
- `offset` : utilisé pour les accès en lecture et écriture pour connaître le point de départ.
- `file` et `dir` : informations supplémentaires pour gérer des fichiers et répertoires qui existent physiquement sur un support de masse (pas implémenté ici).

– `dev` : pointeur vers le descripteur du pilote de périphérique qui implémente les accès au matériel : une structure **Device**.

---

typedef struct *_Device Device*;

struct *_Device* {

char	*name*[*MAX_DEV_NAME_LEN*];	/* name of device */
int	*refcnt*;	/* reference count */
Semaphore *	*mutex*;	/* mutex semaphore */
Semaphore *	*sem_read*;	/* blocking read semaphore */
Semaphore *	*sem_write*;	/* blocking write semaphore */
int	(**init*)(*Device *dev*);	/* device initialization on OS init */
int	(**open*)(*FileObject *f*);	/* open device */
int	(**close*)(*FileObject *f*);	/* close device */
int	(**read*)(*FileObject *f*, **void** **buf*, *size_t len*);	/* read from device method */
int	(**write*)(*FileObject *f*, **void** **buf*, *size_t len*);	/* write to device method */
int	(**ioctl*)(*FileObject *f*, **int** *op*, **void** ***data*);	/* set/get device parameters */

};

Chaque objet Device est associé à un périphérique physique et contient des méthodes qui seront utilisées pour accéder à ce périphérique. On peut ainsi utiliser des protocoles de communication différents pour lire des données, par exemple, provenant du port série et provenant d'une carte SD.

A l'initialisation de l'OS (avant l'appel de la fonction main) toutes les méthodes `init` des périphériques (si elles sont définies) sont appelées.

a. Compléter dans `device/vfs.c` la fonction `open`. Cette fonction renvoie le *descripteur de fichier* (l'index de l'entrée) associé à la structure *FileObject* qui représente le fichier pour l'OS. Le descripteur de fichier est obtenu à partir de la table (`opened_fds`) des descripteurs de fichiers ouverts. On recherche le premier descripteur de fichier libre (entrée nulle) dans la table. Il faut alors allouer et initialiser la structure `FileObject`.

Puis, on traite le cas particulier du répertoire `/dev` qui est un répertoire virtuel, qui sera visible dans l'arborescence sans exister physiquement sur un support de masse : on ajoute, pour le champ `flags` l'attribut `F_IS_DEVDIR` et on renvoie le descripteur de fichier.

Si le nom passé en paramètre n'est pas `/dev`, il faut déterminer à quel **device** physique il faut s'adresser pour réaliser les opérations d'entrées-sorties. Tous les *devices* supportés (structures `Device`) sont enregistrés dans le tableau `device_table` (défini dans `device/target.c`). La fonction `dev_lookup` (dans `device/vfs.c`) permet de parcourir cette table à la recherche d'un nom de périphérique (ex, `/dev/dev_test`) et renvoie un pointeur sur la structure `Device` associée. Il faut initialiser le champ `dev` avec la valeur renvoyée par `dev_lookup`, puis, on exécute la méthode `open` (qui renvoie un booléen 0/1) *si elle existe*. Si tout s'est bien passé, on renvoie le descripteur de fichier, sinon -1 après avoir libéré les ressources.

b. Compléter la fonction `close` qui libère les ressources après exécution de la méthode du device associé.

c. Compléter les fonctions `read`, `write` et `ioctl` qui se contentent d'appeler la méthode du device associé.

3.2. Périphérique de test

MAIN_EX8

Tester le fonctionnement de la couche VFS à l'aide du code d'exemple. Analyser le code de `dev_test` défini dans `device/target.c`. Justifier l'utilisation du mutex.

3.3. Périphérique simple (sans interruption)

MAIN_EX9

a. Implémenter le driver pour contrôler les leds : objet `dev_leds` (dans `device/target.c`). Compléter la fonction `dev_write_leds` qui permet de modifier l'état des leds en fonction de la valeur du paramètre fourni dans le buffer. Pour modifier l'état de la led, on utilisera la fonction `leds(uint32_t val)` définies dans `device/target.c` qui permet de contôler la led RGB (1 bit par led en partant des poids faibles).

b. Tester le fonctionnement de votre driver avec le programme d'exemple.

3.4. Périphérique sur interruption

MAIN_EX10

Principe de l'attente d'évènements asynchrones : une tâche qui attend un évènement asynchrone (par l'appel `read` sur le fichier de périphérique correspondant) doit être bloquée. Lorsque l'évènement intervient, la routine d'interruption du coupleur doit permettre de débloquer la tâche.

Remarque : la priorité NVIC associée aux coupleurs périphériques doit être différente de 0 (valeur réservée pour les SVC). Dans ce cas, il est possible de faire des appels systèmes à partir d'une routine d'interruption.

a. Compléter, dans le fichier `device/target.c`, le device `dev_swuser` qui représente le bouton USER de la carte et les fonctions relatives à son utilisation.

b. Tester avec le programme d'exemple.

3.5. En autonomie maintenant !

On demande de rajouter un/des périphérique(s) de votre choix (une console série (facile), le support du système de fichier sur la carte SD (plus long), …), ainsi que le code de test.