Projet OS (1) : développement du noyau

1. Introduction

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

Le logiciel est scindé en deux parties :

- Le **code applicatif** qui s'exécute en mode **USER**.
  - 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, accès au matériel, …) se fait via les **appels systèmes**.
  - Les interruptions sont autorisées.
  - Lors de l'appel de la fonction `main`, le processeur se trouve en mode **USER** et les interruptions sont inhibées. Une initialisation minimale du système a été effectuée.
- Le **système d'exploitation** composé :
  - du **noyau** qui s'exécute en mode privilégié (**SUPERVISOR/IRQ**)
    - Il implémente le code et met en place les structures de données nécessaires pour offrir un certain nombre de services à la couche applicative. Ces services sont invoqués par l'application utilisateur via des appels systèmes. Les appels systèmes sont des opérations atomiques (non interruptibles).
    - Il gère la **commutation de tâches** et décide de la tâche applicative qui peut utiliser le processeur à un instant donné.
    - Il gère les **interruptions** et peut provoquer une commutation des tâches applicatives en réponse au traitement d'une interruption.
  - d'une **couche driver** qui s'exécute en mode **USER** 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.

Le système sera construit de manière progressive : ce premier labo (4 UC) mets en place le noyau (en fait, plutôt un micro-noyau …) et aborde les points suivants :

- Mise en place de la couche d'appels systèmes.
- Création d'une première tâche, avec l'appel système sous-jascent et mise en route de cette première tâche.
- Mise en place de la commutation entre tâches sur la base d'une interruption générée par l'appui d'un bouton poussoir dans un premier temps, puis générée de manière régulière par un timer par la suite.
- Mise en place de Mutex pour assurer l'exclusion mutuelle aux ressources communes aux différentes tâches, et plus généralement permettre la synchronisation entre tâches (sémaphores).
- Mise en place de temporisations pour les tâches. La ligne de conduite sous-jascente est qu'il ne faut permettre à une tâche de s'exécuter (d'utiliser le CPU) que dans la mesure où elle a "*quelque chose d'utile*" à faire. Ainsi, une tâche qui doit être endormie pendant un certain temps ne devrait pas être réactivée avant que la durée en question ne soit écoulée.

Le répertoire du projet est organisé de la manière suivante :

```
/répertoire racine du projet/
 +- Makefile                        : fichier de construction du projet
 +- include/
 |  +- typedefs.h                   : redéfinition de types
 |  +- oslib.h                      : liste des appels systèmes supportés par le noyau
 |
 +- user/
 |  +- main.c                       : code utilisateur (tests, application finale).
 |
 |                          C O D E   U T I L I S A T E U R
 | ------------------------------------------------------------------------------------
 |                                C O D E   N O Y A U
 |
 +- kernel/ (code portable, exécution en mode SUPERVISEUR)
 |  +- syscall_dispatch.c : dispatcher d'appels systèmes
 |  +- list.[ch]    : fonction de manipulation de listes circulaires
 |  +- kernel.[ch] : implémentation du noyau
 |
 +- oslib/ (exécution en mode USER)
    +- kernellib.c                  : fichier contenant la couche d'appels systèmes côté
    |                                 utilisateur
    +- drivers/ (code spécifique à l'architecture matérielle)
       +- target.[ch]               : interface générique avec le noyau (init + drivers
       |                              matériels)
       +- config/sram.lds           : config pour le placement mémoire
       +- startup/board_cstartup.s  : code startup + sauvegarde/restitution de contexte
       |                              pour les modes SVC/IRQ
       +- include/                  : interface pour les accès au matériel à bas niveau
       +- lib/                      : implémentation pour les accès au matériel de bas
                                      niveau
```

Le code source du labo, ainsi que la documentation du microcontrôleur sont disponibles sur Moodle.

2. Les appels systèmes

- Du côté utilisateur, les appels systèmes sont vus comme des appels de fonctions. La partie *utilisateur* ne fait que l'appel de l'instruction **swi** avec le numéro de l'appel système.

  Exemple : l'appel système 12 de prototype

  int *read*(**int** *fd*, **void** **buf*, *size_t len*)
  {
  int *size_or_err*;
  asm(*"swi 12\n\tmov %0, r0"* : *"=r"* (*size_or_err*));
  return *size_or_err*;
  }

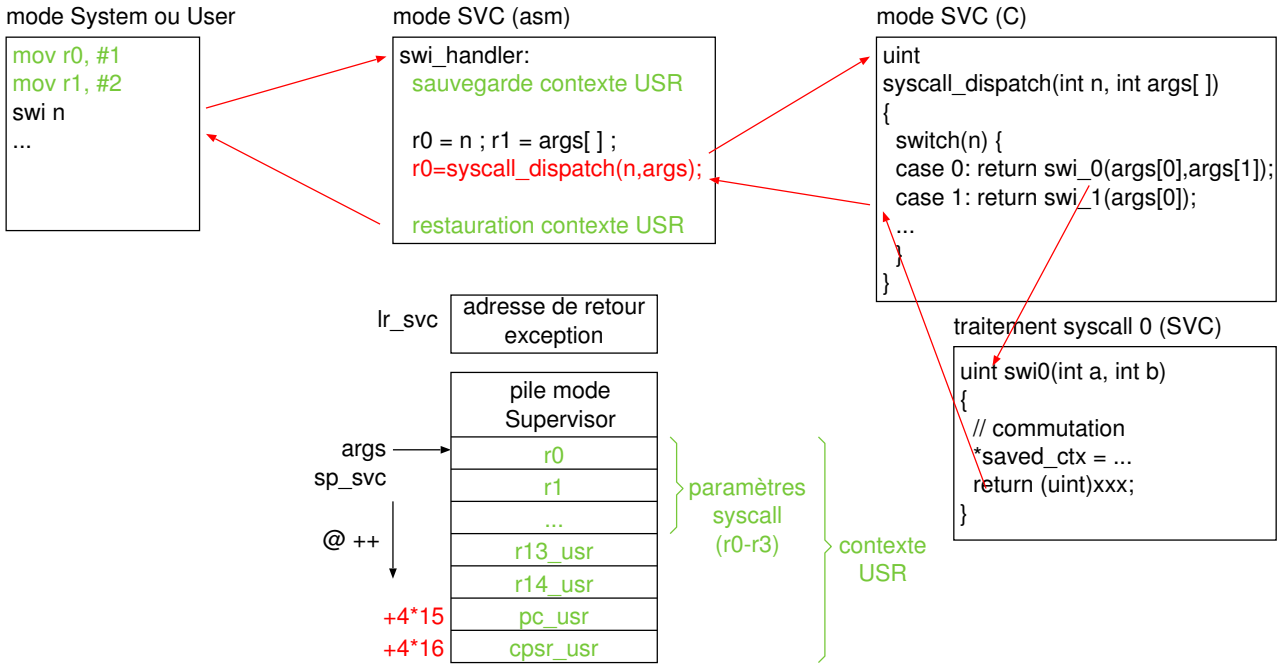
  Conformément au standard AAPCS d'ARM, l'appel système passe ses paramètres par les registres `r0` (premier paramètre), `r1` et `r2` (4 paramètres au maximum) et renvoie la valeur de retour par le registre `r0`. L'instruction `mov` associée oblige ici copier le résultat `r0` dans la variable `size_or_err`.

- L'implémentation est réalisée au niveau du code de l'OS.

  Lors de l'exécution de l'instruction swi, le processeur bascule en mode SUPERVISEUR (SVC) et va chercher l'instruction à l'adresse `0x00000008` (vecteur d'exception logicielle) qui permet d'appeler la fonction assembleur `swiHandler` (définie dans `oslib/drivers/startup/board_cstartup.s`). Cette fonction réalise une sauvegarde complète du contexte du mode USR sur la pile SVC et appelle la fonction C `syscall_dispatch` (définie dans `kernel/syscall_dispatch.c`) avec pour paramètres le numéro de l'appel système et un tableau contenant les paramètres qui ont été passés à l'appel système. Cette fonction a pour prototype :

  /* syscall_dispatch
  * dispatch syscalls
  * n : syscall number
  * args[] : array of the parameters (4 max)
  */
  int32 syscall_dispatch(*uint32 n*, *uint32 args*[]);

  Lors du retour de l'appel système, le contexte USR est rétabli avant de rebasculer vers le mode USR.



a. Compléter les fichiers `oslib/kernellib.c` et `kernel/syscall_dispatch.c` pour définir et implémenter l'appel système `test_add` qui réalise une simple addition des valeurs passées en paramètres.

b. Tester. Observer en particulier le changement de mode du processeur et la manière dont les paramètres sont passés et la valeur de retour renvoyée.

c. Compléter les fichiers `oslib/kernellib.c` et `kernel/syscall_dispatch.c` pour que les autres appels système soient correctement appelés. Du côté noyau, l'implémentation de l'appel système est définie dans les fichiers `kernel/kernel.[ch]`. Les noms des fonctions qui implémentent un appel système 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
malloc	:	1
free	:	2
os_start	:	3
task_new	:	4
task_id	:	5
task_wait	:	6
sem_new	:	7
sem_p	:	8
sem_v	:	9

d. Tester la mécanique d'appel pour quelques appels systèmes.

e. Comment est obtenue l'atomicité des appels systèmes ?

3. Création et lancement d'un processus

Une **tâche** 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 par :

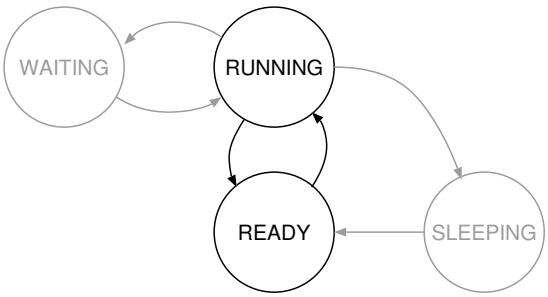
```
typedef struct _Task {
    struct _Task *  prev;       // prev task
    struct _Task *  next;       // next task
    uint32          id;         // identifier
    TaskState       status;     // task state : running, ready, ...
    uint8 *         stack_base; // bottom stack address
    Context         ctx;        // USR registers : r[0] -> r0, ... r[15] (pc), cpsr
    int32           delay;      // wait delay
} Task;
```

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

Les différents champs de la structure sont les suivants :

– `id` est un identificateur 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()`.

– `stack_base` : adresse du bloc alloué dynamiquement et servant de pile au code de la tâche.

– `ctx` : contexte du processus = valeur des registres lorsque la tâche n'est pas en train de s'exécuter. Il s'agit d'une structure définie dans `kernel/kernel.h`

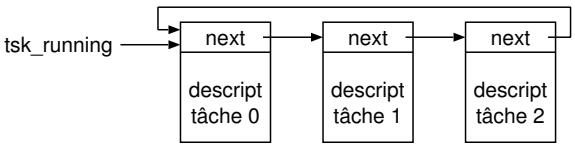
```
typedef struct _Context {
   uint32   r[16];
   uint32   cpsr;
} Context;
```

∗ r[] représente les registres utilisés par le code de la tâche en mode UTILISATEUR, avec en particulier : le pointeur de pile r13, le registre de lien r14 et le compteur programme r15.

* cpsr représente le contenu du regsitre d'état de la tâche en mode UTILISATEUR.

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

a. Compléter l'appel système `task_new` qui permet d'allouer dynamiquement un descripteur pour la tâche créée, qui 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`. La fonction renvoie l'identificateur de tâche `id` ou -1 si la structure n'a pu être allouée. En clair quand on a créé 3 tâches, on doit avoir :



On utilisera l'allocateur mémoire intégré au noyau dans les fichiers `kernel/alloc.[ch]`. Lorsqu'on est en mode USR, cet allocateur sera manipulé par les fonctions standard (`malloc` / `free`). Par contre, en mode noyau, il est nécessaire d'avoir recours aux fonctions `sys_alloc` / `sys_free`.

On utilisera également les fonctions de manipulation de listes circulaires des fichiers `kernel/list.[ch]` (un exemple d'utilisation est fourni).

Le contexte stocké dans le descripteur de tâche identifie l'état de la tâche lors de sa création. Il doit permettre de lancer le code dont l'adresse est fournie en paramètre en mode USR et en permettant au processeur d'interrompre la tâche pour traiter une interruption (IRQ). Un rappel des registres généraux et du registre d'état du processeur ARM est fourni en annexe.

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

c. Lorsqu'un appel système est réalisé ou qu'une interruption est générée, le contexte d'exécution en mode USR (la structure Context) est sauvegardé sur la pile SVC et la variable globale `saved_ctx` (définie dans `kernel/kernel.c`) est initialisée (dans `oslib/drivers/startup/board_cstartup.s`) avec l'adresse de la structure Context sauvegardée sur la pile. Sachant que le contexte présent sur la pile SVC au moment du retour de l'appel système sert de nouveau contexte USR, compléter l'appel système `os_start` qui permet de démarrer la première tâche (pointée par `tsk_running`).

d. Tester le code et vérifier qu'on peut effectivement démarrer la première tâche (mais pas les autres pour le moment).

4. 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*. Une telle commutation s'effectue en mode SVC et peut avoir lieu *lors d'un appel système* ou *lors du traitement d'une interruption*. Pour réaliser une commutation entre deux tâches, il suffit de sauvegarder le contexte de la tâche interrompue (dans son descripteur de tâche) et de le remplacer par celui de la tâche qui doit prendre la main.

a. *Algorithme du tourniquet (Round Robin)*

Compléter la fonction `sys_round_robin` (dans `kernel/kernel.c`) qui permet d'attribuer le processeur à la tâche suivante dans la liste des tâches prêtes.

b. *Commutation "manuelle"*

On utilise l'interruption générée par le bouton P0.14 de la carte ARM pour provoquer la commutation de tâches. Lorsque le bouton est appuyé, la fonction de rappel `on_button_cb` (définie dans `oslib/drivers/target.c`) est appelée par la routine de bas niveau de traitement de l'interruption (le driver) pour provoquer la commutation vers la tâche suivante dans la liste des tâches prêtes.

La configuration du coupleur périphérique est réalisée dans la fonction `dev_init` de `oslib/drivers/target.c` appelée avant l'exécution de la fonction `main`. Les acquittements d'interruption auprès du coupleur et du contrôleur d'interruptions sont pris en charge par le code du driver (`os/drivers/lib/eint1.c`) et le code assembleur (`os/drivers/startup/board_cstartup.s`).

Le programme de chaque tâche écrit son numéro d'identificateur sur le terminal. Vérifier le fonctionnement. On pourra utiliser la fonction `list_display(tsk_running)` pour visualiser l'ordre des tâches dans la liste des tâches prêtes au moment de chaque commutation.

c. *Commutation "automatique"*

On utilise l'interruption périodique (définie par la macro `SYS_TICK` (10 ms) définie dans `oslib/drivers/target.h`) en provenance du _TIMER0 du microcontrôleur pour provoquer la commutation de tâches. Compléter le code de la fonction `on_timeout_cb` (définie dans `oslib/drivers/target.c`) appelée à chaque fois que l'interruption est générée. Il faut également compléter l'appel système `os_start` afin de démarrer le timer (voir le prototype de la fonction `sys_timer_start` dans `oslib/drivers/target.h`) avant de lancer la première tâche. La configuration du Timer est réalisée à l'initialisation du système (fonction `dev_init`). L'acquittement de l'interruption est pris en charge par le driver.

Observer le comportement de l'affichage sur la liaison série. Expliquer.

d. Modifier la variable `PROJ` du `Makefile` pour utiliser le fichier *usr/tp1_ex2.c*. Visualiser le fonctionnement. Justifier en comparaison avec le code précédent.

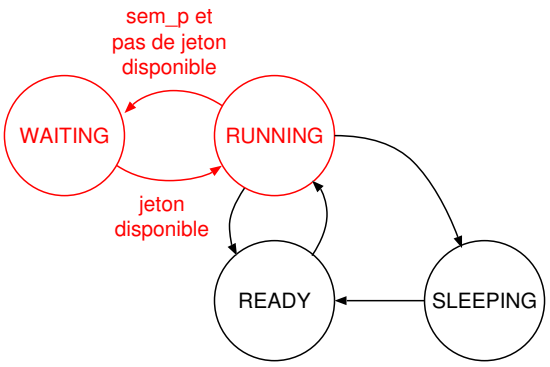
5. Les sémaphores

Un **sémaphore** est un compteur associé à une liste de tâches bloquées devant. Le compteur indique le nombre de jetons disponibles. On implémente 3 appels systèmes (primitives) pour manipuler les sémaphores :

`sem_new` : création du sémaphore et initialisation du compteur.

`sem_p` : la tâche qui exécute l'appel système demande un jeton et devient bloquée s'il n'y en a plus de disponible.

`sem_v` : la tâche qui exécute l'appel système remet un jeton, ce qui peut éventuellement débloquer une tâche en attente d'un jeton.



Du point de vue de l'utilisateur, le sémaphore est un objet opaque qu'on ne peut manipuler qu'au travers des trois appels systèmes précédents. Pour le concepteur de l'OS (vous :-), le sémaphore est la structure C suivante (décrite dans le fichier `kernel/kernel.c`) :

```
typedef struct _Semaphore {
        int32   count;
        Task * waiting;
} Semaphore;
```

Les prototypes des appels systèmes sont :

```
/* sem_new
 *   create a semaphore
 *   init    : initial value
 */
Semaphore * sem_new(int32 init);

/* sem_p
 *   take a semaphore
 */
void sem_p(Semaphore * sem);

/* sem_v
 *   release a semaphore
 */
void sem_v(Semaphore * sem);
```

L'implémetation 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
`sem_v`	:	incrémenter le compteur s'il y a des tâches en attente alors débloquer la première tâche en attente et lui donner le processeur. finsi

a. Coder l'implémentation des appels systèmes `sem_new`, `sem_p` et `sem_v`. La vérification du fonctionnement correct des sémaphores pourra se faire en utilisant l'application utilisateur contenue dans le fichier `user/tp1_ex3.c`.

```
#include <stdlib.h>
#include <oslib.h>
#include <uart.h>
#include <io.h>

Semaphore * sem=NULL;
int cpt=0;

void tache1()
{
        while (1) {
                sem_p(sem);
                uart_printf(_UART0, "tache 1 : cpt = %d\n",++cpt);
        }
}

void tache2()
{
        while (1) {
                volatile int k;
                if (button0()) sem_v(sem);
                for (k=0;k<1000;k++) ;
        }
}

int main()
{
        sem = sem_new(0);

        task_new(tache1,1024);
        task_new(tache2,1024);

        os_start();
        return 0;
}
```

L'application met en place deux taches :

- la tâche 2 attend l'appui sur le bouton 0 (P0.16) et envoie un jeton au sémaphore *sem*,
- 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. Vérifier à l'aide du programme utilisateur `usr/tp1_ex4.c` que la mise en place de l'exclusion mutuelle sur les accès aux périphériques permet de corriger les effets observés lors de la mise en place de la commutation automatique.

6. Gestion du temps

Pour temporiser le fonctionnement d'une tâche, il est envisageable de la ralentir en lui faisant exécuter du 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 imposée par le timer (définie par la macro `SYS_TICK`=10 ms dans le fichier `oslib/drivers/target.h`).

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 tête) dans la liste des tâches prêtes.

a. Implémenter l'appel système `task_wait` et vérifier son fonctionnement à l'aide du programme utilisateur `usr/tp1_ex5.c`.

b. Vérifier le fonctionnement sur le programme utilisateur `usr/tp1_ex6.c`.

7. Pour aller plus loin …

Un certain nombre d'améliorations peuvent être apportée au niveau du noyau

a. Modifier le `Makefile` pour utiliser le fichier *usr/tp1_ex7.c*. 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.

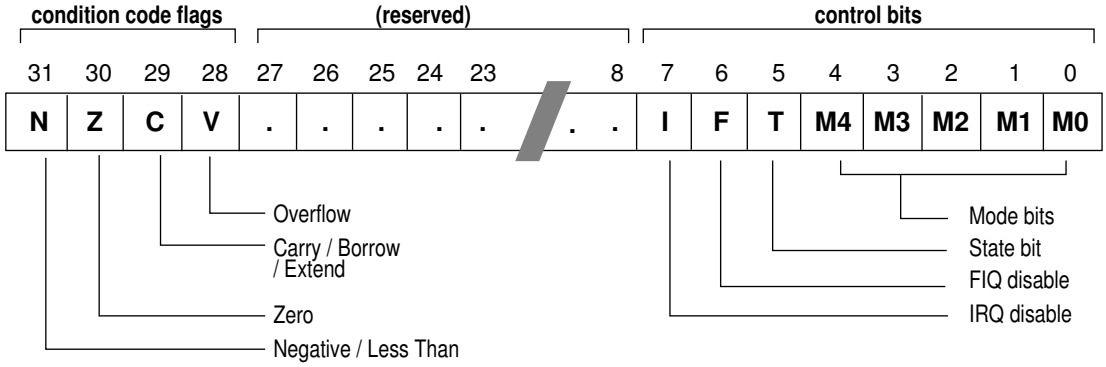
   Proposer et coder une solution permettant de reprendre la main au niveau de l'OS pour détruire proprement la tâche qui s'arrête et donner la main à une autre. On pourra introduire un appel système `task_kill` (appel système 6) qui "*détruit*" la tâche qui exécute cet appel système.

b. A ce stade, tous les problèmes classiques (producteur-consommateur, lecteurs-rédacteurs, philosophes, …) devraient pouvoir être traités …

c. Il existe d'autres systèmes de synchronisation que les sémaphores, par exemple, les boîtes aux lettres (mailbox) qui permettent d'échanger des messages entre tâches. On peut implémenter les appels systèmes : `mb_new` / `mb_send` / `mb_recv`.

8. Annexe : rappels sur l'architecture de programmation du processeur ARM

- **Le registre d'état** :
  - Le registre d'état CPSR comprend les flags de l'UAL (NZCV), les bits d'autorisation de prise en compte des interruptions FIQ (non utilisé ici[1]) ou IRQ (utilisé [0]), le mode d'exécution (ARM[0]/Thumb[1]), ainsi que le mode du processeur.



  - Un mode du processeur est dit privilégié s'il permet un accès complet, en lecture et en écriture, au registre d'état CPSR (en particulier à l'octet de poids faible). Tous les modes de l'ARM sont privilégiés sauf le mode User.
  - En mode utilisateur on peut lire le registre d'état. Mais seul l'octet des drapeaux UAL (CPSR_f) est modifiable.
- **Les modes de fonctionnement** sont encodés de la manière suivante :

M[4:0] (CPSR)	Mode	Commentaires
10000	User	mode de fonctionnement du logiciel utilisateur.
10001	FIQ	mode dans lequel est traitée une exception matérielle déclenchée par le signal `nFIQ`.
10010	IRQ	mode dans lequel est traitée une exception matérielle déclenchée par le signal `nIRQ`.
10011	SVC	(superviseur) mode dans lequel sont traitées les exceptions déclenchées par le signale `nRESET` et l'instruction SWI.
10111	Abort	mode dans lequel sont traitées les exceptions *Data & Prefetch Abort*.
11011	Undefined	mode dans lequel est traitée une exception *Undefined Instruction*.
11111	System	mode réservé (accès privilégié, mais contexte utilisateur).

- Certains registres sont dupliqués et sont donc propres à chacun des modes (R13 et R14).

ARM State General Registers and Program Counter

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

ARM State Program Status Registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

- **La table des vecteurs d'exceptions** contient les instructions exécutées lorsqu'une exception intervient.

Adresse	Exception	Mode du µP
0x00000000	Reset	Superviseur
0x00000004	Undefined Instruction	Undefined
0x00000008	Software Interrupt	Superviseur
0x0000000C	Prefetch abort	Abort
0x00000010	Data abort	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

- La table des vecteurs est remappée en SRAM à l'adresse 0x40000000.
- Les vecteurs doivent contenir chacune une instruction de saut vers la routine de traitement d'exception appropriée.
- Afin de permettre une localisation quelconque des routines d'exception dans l'espace adressable, la table des vecteurs doit comprendre deux parties, selon le canevas ci-dessous :
  1. Les instructions de saut. Le saut est effectué en chargeant l'adresse de la routine d'exception dans le PC.
  2. Les adresses des routines d'exception.

Voir le fichier `oslib/drivers/startup/board_cstartup.s`

- **Mapping mémoire**
  - FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 512K
  - SRAM1 (rwx) : ORIGIN = 0x40000000, LENGTH = 32K
  - SRAM2 (rwx) : ORIGIN = 0x7FD00000, LENGTH = 8K

  Pour plus de détails : voir le document UM10139.pdf sur Moodle

- **Edition de lien et sections**

  Le script de configuration de l'édition de lien se trouve dans le fichier

  ```
  oslib/drivers/config/sram.lds
  ```

  Après compilation, l'entrée dans un terminal de la commande :

  ```
  arm-none-eabi-objdump -h main.elf
  ```

  permet de connaître les adresses à partir desquelles se trouvent les sections :

  * `.vectors` : vecteurs d'interruption de l'ARM,
  * `.text` : code exécutable,
  * `.rodata` : zone de constantes,
  * `.data` : zone de variables globales initialisées,
  * `.bss` : zone de variables globales non initialisées,
  * `.stack` : zone utilisée pour les piles des différents modes de fonctionnement (ne concerne pas l'emplacement des piles de tâches),

  Le tas (heap) démarre après la zone de pile jusqu'à la fin de la SRAM

  Un résumé est généré par la commande :

  ```
  arm-none-eabi-size main.elf
  ```

  Ces deux commandes sont exécutées à chaque fois qu'un nouvel exécutable est généré.

  L'emplacement des variables, fonctions, … peut être obtenu via le fichier `nomprojet.map` généré à la racine du répertoire projet après chaque édition de lien.

  Certaines zones sont situées à deux emplacements donnés par les VMA (Virtual Memory Address) et LMA (Load Memory Adress) (par exemple, la section `.data`). Les adresses VMA sont les adresses à partir desquelles les données sont manipulées par le programme (en RAM). Les adresses LMA sont des adresses à partir desquelles on va trouver, par exemple, les valeurs initiales (typiquement en mémoire Flash – pas sur notre config de développement) des variables globales qui seront copiées vers la zone d'utilisation de ces variables en RAM.