Introduction

Ce document décrit comment pourrait être implémentée la planification de tâches dans un système d'exploitation préemptif en utilisant un cœur ARM Cortex M4.

Le Système

Supposons un programme dans lequel trois tâches peuvent être exécutées par le système d'exploitation préemptif comme suit :

Main	Tâches	Interruption du SysTick

//char PileMain[256];
 	
void main(void)
{
__disable_interrupt();
initSystick();
initPileT1();
initPileT2();
initPileT3();
PileActive = &pPileMain;
__enable_interrupt();
 	
while(1)
{};
}
	char PileT1[256];
	char PileT2[256];
	char PileT3[256];
	
	void T***x***(void)
	{
	initT***x***();
	while(1)
	{
	ExecuteT***x***();
	}
	}
	
	Où ***x*** peut valoir 1, 2 ou 3…
		char** PileActive;
		char *pPileMain;
		char *pPileT1;
		char *pPileT2;
		char *pPileT3;
		
		void SysTick_Handler(void)
		{
		SauvegardeContexte();
		PileActive = ChoixTacheAExecuter();
		ChargementContexte();
		}

Dans ce système simple, chaque tâche est exécutée à tour de rôle par le système d'exploitation : la fonction ChoixTacheAExecuter (); dans l'interruption du SystTick (l'interruption du système d'exploitation) est appelée périodiquement et change la tâche exécutée.

L'interruption du SysTick est une interruption de Timer programmé en mode AutoReload : elle survient à intervalle régulier dans le temps.

Notez que les tâches sont admises dans la mémoire de manière statique, c'est-à-dire par le compilateur. Habituellement, la mémoire allouée aux tâches ou processus est allouée de manière dynamique, par le système d'exploitation. Par ailleurs, une structure de données, le PCB (Process Control Block), contient habituellement un pointeur sur la pile de chaque tâche en mémoire pour permettre de gérer les tâches dynamiquement. Dans cet exemple, le PCB est représenté par *char *pPileT**x**;*.

Implémentation du changement de contexte avec 1 pointeur de pile

L'implémentation proposée ici sauvegarde le contexte de chaque tâche sur sa propre pile : les registres et les drapeaux propres à T1, T2 ou T3 sont sauvegardées sur PileT1, PileT2 ou pileT3. Pour passer de la tâche T1 à la tâche T2 par exemple, le système d'exploitation doit empiler toutes les informations reliées à T1 sur pile T1, puis il doit dépiler les informations de pile T2 pour revenir à la tâche T2.

Le changement de contexte est fait en manipulant les piles. Les adresses de retour vers T1, T2 et T3 sont sauvegardées sur la pile de chaque tâche.

Pour comprendre comment implémenter le système, regardons ce qui se passe dans le main après le *__enable_interrupt();*:

- En raison du while(1){}, le microprocesseur tourne en rond jusqu'à ce que l'interruption du SysTick ne survienne.
- Lors de l'interruption du SysTick, le cœur ARM Cortex M4 empile automatiquement des registres sur la pile du main (la mémoire allouée pour la pile du main est allouée par le compilateur) : le microprocesseur sauvegarde automatiquement R0 à R3, R12, les drapeaux (Program Status Register), le registre de lien et ***l'adresse de retour*** vers le main (vers le while(1){}!!!).

Lorsque la première instruction de la routine traitant l'interruption est exécutée, la pile du main ressemble à ceci :

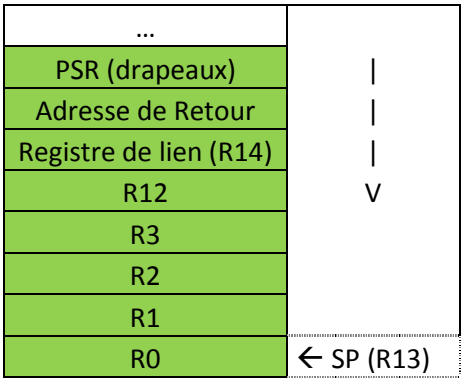


Figure 1- Pile du Main après l'entrée dans l'interruption

Notez qu'il s'agit d'une pile descendante : lors d'un PUSH, le pointeur de pile, SP, est décrémenté, puis la valeur est mise sur la pile (« full descending »). Le pointeur de pile indique la dernière entrée valide.

Habituellement, la première instruction exécutée dans void SysTick_Handler(void) n'est pas l'appel vers la routine *SauvegardeContexte();* : le compilateur met sur la pile tous les registres qui sont utilisés dans la routine d'interruption (et qui ne sont pas sauvegardés automatiquement par le microprocesseur). De plus, le compilateur fait habituellement un PUSH du registre de lien parce que ce dernier est changé au début de l'interruption (automatiquement par le microprocesseur): il contient EXC_RETURN (execution return), c'est-à-dire la valeur qu'il faut mettre dans PC (R15) pour retourner correctement de l'interruption. Ici, « correctement » signifie à l'endroit de départ. La pile, juste avant l'appel de SauvegardeContexte(); (après le code du compilateur et avant le code du programmeur) ressemble à :

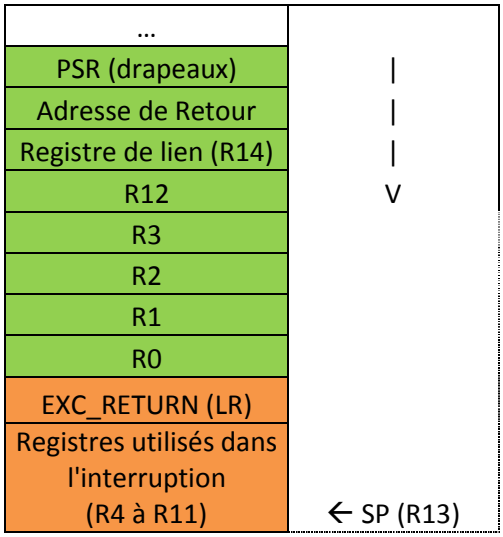


Figure 2- Pile du Main après l'entrée dans l'interruption et l'exécution du code « caché » du compilateur

Avec ma version d'IAR, la première instruction exécutée dans une routine d'interruption est un PUSH LR. Cette instruction est suivie de PUSH R7!

Le microprocesseur exécute maintenant le code du programmeur :

SauvegardeContexte();
ChoixTacheAExecuter *();*
ChargeContexte()

Sauvegarde du Contexte

Pour sauvegarder le contexte en entier, il ne manque que la sauvegarde de R4 à R11, tous les autres registres étant sauvegardés automatiquement par le microprocesseur (voir Figures précédentes). La sauvegarde du contexte, en assembleur, est conceptuellement simple :

```
SauvegardeContexte :   ;Concept seulement!!!
        stmdb sp!, {r4-r11}      ;PUSH R4 à R11
         BX    lr      ;Return
```

Choix de la Tâche À Exécuter

Il vous faudra déterminer si T1, T2 ou T3 doit être exécuté dans le prochain quantum, c'est-à-dire entre cette interruption du système d'exploitation et la suivante.

À cette étape, les piles de toutes les tâches doivent être prêtes pour l'exécution de la tâche :

…	…	…	…
PSR (drapeaux)	PSR (drapeaux)	PSR (drapeaux)	PSR (drapeaux)
Adr. Ret -- ***MAIN***	Adr. Ret -- ***T1***	Adr. Ret -- ***T2***	Adr. Ret -- ***T3***
Registre de lien (R14)	Registre de lien (R14)	Registre de lien (R14)	Registre de lien (R14)
R12	R12	R12	R12
R3	R3	R3	R3
R2	R2	R2	R2
R1	R1	R1	R1
R0	R0	R0	R0
EXEC_RETURN (LR)	EXEC_RETURN (LR)	EXEC_RETURN (LR)	EXEC_RETURN (LR)
Registres (R4 à R11) utilisés dans l'interruption	Registres (R4 à R11) utilisés dans l'interruption	Registres (R4 à R11) utilisés dans l'interruption	Registres (R4 à R11) utilisés dans l'interruption
Registres R4 à R11	Registres R4 à R11	Registres R4 à R11	Registres R4 à R11

Figure 3- Piles du système lorsqu'on détermine la tâche à exécuter

Cela indique comment initialiser les piles de chaque tâche! Les fonctions *initPileT**x**()* doivent préparer les piles de chaque tâche pour que la tâche soit exécutée lors du retour de l'interruption du système d'exploitation. La valeur initiale de l'Adresse de Retour est simplement l'adresse de la fonction décrivant la tâche (l'adresse de *void T1(void)* par exemple). La valeur initiale des autres éléments de la pile devrait être similaire à celle des éléments de la pile du main.

Pour choisir la tâche à exécuter, il suffit de changer la pile active!

Chargement du Contexte

Pour exécuter une tâche et rétablir son contexte, il faut simplement dépiler la pile de ce contexte. Cela se fait en plusieurs étapes. Dans un premier temps, la fonction ChargementContexte() dépile R4 à R11. Ensuite, du code « caché » du compilateur dépile les registres R4 à R11 utilisés dans l'interruption et dépile EXEC_RETURN dans PC. Cette instruction (retour d'interruption) entraîne le microcontrôleur à dépiler automatiquement les registres R0 à R3, R12, R14 et les drapeaux. De plus, le microcontrôleur dépilera l'adresse de retour et ramènera l'exécution du code dans la tâche désirée. La fonction de chargement de contexte, en assembleur, est conceptuellement simple :

```
ChargementContexte:  ;Concept seulement!!!
        ldmia sp!, {r4-r11}      ;POP R4 à R11
        BX    lr     ;Return
```

Pointeurs de pile et exécution de tâche

Lorsque commence l'exécution d'une tâche, la pile de cette tâche est vide : l'information nécessaire au système d'exploitation pour exécuter la tâche a été dépilée. Pendant l'exécution de la tâche, la pile de la tâche se remplira. Par exemple, si la fonction 1 est appelée dans une tâche, l'adresse de retour de la fonction 1 sera éventuellement mise sur la pile (en considérant que le registre de lien est empilé systématiquement.

Si on considère le code suivant pour la tâche 1:

Tâche 1	Fonction 1	Fonction 2

void T1(void)
{
initT1();
while(1)
{
  Fonction1();
  if(event())
  {
  Fonction3();
  }
}}
	void Fonction1(void)
	{
	PUSH LR;
	PUSH R0;
	PUSH R1;
	Fonction2();
	Fonction4();
	Fonction5();
	POP R1;
	POP R0;
	POP LR;}
		void Fonction2(void)
		{
		int a;
		int b;
		int c;
		
		a = LitADC0();
		b= LitADC1();
		c = CalculeC(a,b);
		//INT du OS ici ?!?
		return;}

Si une interruption du OS survient aléatoirement dans la Fonction 2, à l'endroit marqué par « INT du OS ici », la pile de la tâche 1 ressemblera à ceci, après la sauvegarde du contexte de la tâche 1 :

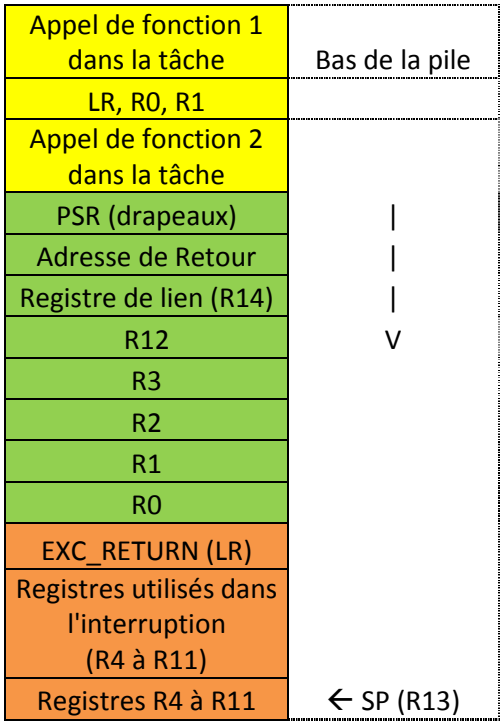


Figure 4- Illustration d'une pile de tâche après quelques exécutions, après la sauvegarde de contexte

À toutes les fois qu'on interrompt une tâche, le pointeur de la pile de la tâche interrompue peut prendre différentes valeurs. Dans l'exemple précédent, la valeur de SP finale, après la sauvegarde de contexte par l'OS, aurait été différente si l'interruption du OS était survenu à un autre moment. Lors de la sauvegarde de contexte, il faudra donc noter la valeur de SP de la tâche active :

SauvegardeContexte :

```
stmdb sp!, {r4-r11}      ;PUSH R4 à R11

 ;Sauvegarde de SP de la tâche active
 ldr r0, = PileActive        ;Met l'adresse de PileActive dans r0
 ldr r1, [r0]  ;Met l'adresse de pPileT1, pPileT2 ou pPileT3 dans r1
 str sp, [r1]  ;Sauvegarde le pointeur de pile de la tâche

 BX    lr     ;Return
```

Pour décider de la tâche à effectuer, ChoixTacheAExecuter doit décider si PileActive pointera sur pPileT1, pPileT2 et pPileT3…

ChargementContexte:

```
;Chargement de SP de la tâche active
 ldr r0, = PileActive        ;Met l'adresse de PileActive dans r0
 ldr r1, [r0]  ;Met l'adresse de pPileT1, pPileT2 ou pPileT3 dans r1
 ldr sp, [r1]  ;Sauvegarde le pointeur de pile de la tâche

ldmia sp!, {r4-r11}      ;POP R4 à R11

BX    lr     ;Return
```

La tâche du « main »

Dans le système ci-dessus, il n'est pas obligatoire de terminer le main par un while(1){}. En effet, toutes les ressources sont disponibles pour traiter le main comme les autres tâches (comme T1, T2 ou T3). Pour continuer l'exécution du main après une interruption du système d'exploitation, il suffit de mettre PileActive = &pPileMain; lorsqu'on détermine la tâche à effectuer…

```
void main(void)
{
…
while(1)
{
  ExecuteTacheDuMain();
}
}
```

Implémentation du changement de contexte avec deux pointeurs de pile

Dans l'exemple précédent, le système comprend les tâches à exécuter et l'interruption du système d'exploitation. Cependant, il ne contient pas les interruptions des périphériques. Dans la plupart des systèmes microprocesseurs (presque tous!), des périphériques déclencheront des interruptions qui seront traitées dans leur propre contexte (dans l'ISR de chaque interruption).

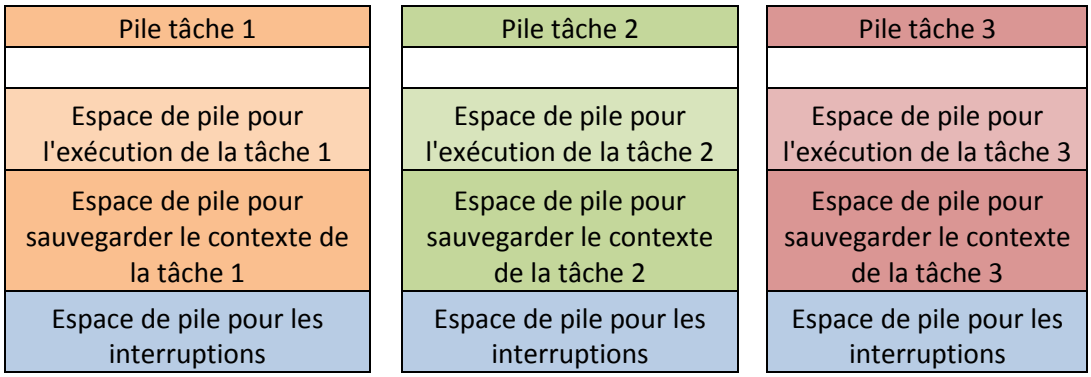
Si on suppose qu'une interruption du UART se produit pendant l'exécution de la tâche 1 par exemple, à la place de l'interruption du OS dans l'exemple précédent, la pile de la tâche 1 ressemblera à ceci :

Empilé dans la tâche	Appel de fonction 1 dans la tâche	Bas de la pile
	LR, R0, R1	
	Appel de fonction 2 dans la tâche	
Empilé par le microprocesseur lors de l'interruption du UART	Drapeaux	
	Adresse de retour (quelque part dans la fonction 2)	
	LR, R12, R0 à R3	<--SP

Si on suppose une interruption d'I2C, de priorité supérieure à celle du UART, qui survient pendant l'interruption du UART, la pile de la tâche ressemblera à ceci :

Empilé dans la tâche	Appel de fonction 1 dans la tâche	Bas de la pile
	LR, R0, R1	
	Appel de fonction 2 dans la tâche	
Empilé par le microprocesseur lors de l'interruption du UART	Drapeaux	
	Adresse de retour (quelque part dans la fonction 2)	
	LR, R12, R0 à R3	<--SP
	Appel de fonction dans l'ISR du UART	
Empilé par le microprocesseur lors de l'interruption du I2C	Drapeaux	
	Adresse de retour (quelque part dans l'ISR du UART)	
	LR, SP, R12, R0 à R3	<--SP

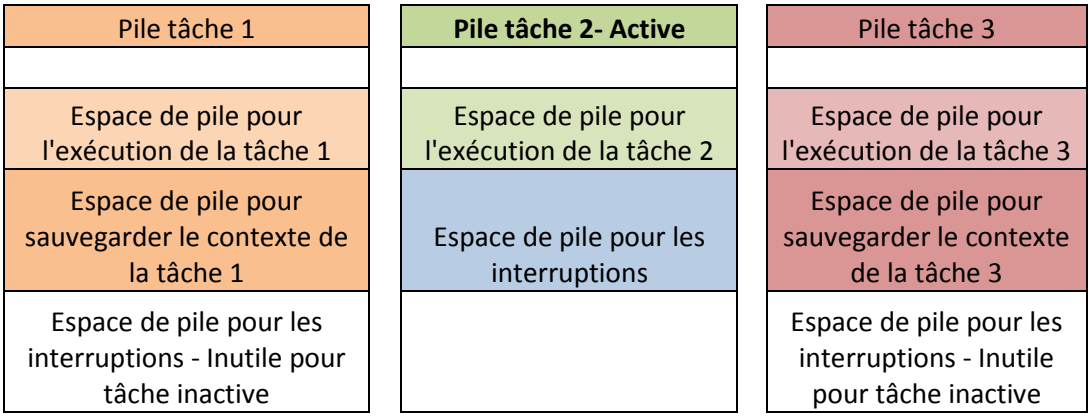
La pile de la tâche doit avoir une taille suffisante pour entreposer les informations survenant lors d'interruptions. Comme les interruptions peuvent survenir pendant l'exécution de n'importe quelle tâche, *les piles de toutes les tâches doivent avoir une taille suffisante pour traiter toutes les interruptions*.



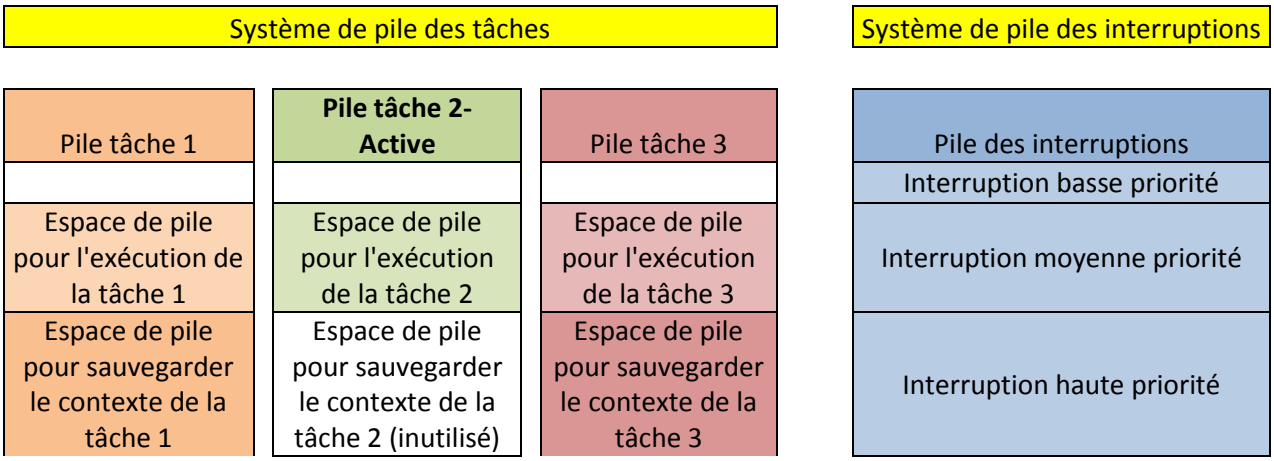
Habituellement, les interruptions des périphériques ont une priorité supérieure à celle du système d'exploitation.

> *Si l'interruption d'un périphérique a une priorité inférieure à celle du OS et si le traitement de cette interruption est interrompu par l'interruption du OS, le traitement de l'interruption sera repris uniquement lorsque l'OS décidera de continuer l'exécution de la tâche qui était active…*

De ce fait, l'espace de pile réservé pour les interruptions ne peut être utilisé que dans une seule tâche à la fois (la tâche active), et jamais lorsque le contexte de la tâche est sauvegardé : comme elles ont une priorité supérieure, les interruptions des périphériques se termineront et libéreront la pile toujours avant un changement de contexte ou une interruption du système d'exploitation. Si on suppose que la tâche 2 est active, les piles des tâches ressembleront à ceci :



De l'espace mémoire RAM est gaspillé : il y a de l'espace pour toutes les interruptions dans toutes les piles, mais un seul espace est utilisé. Pour éviter ce gaspillage de RAM, les concepteurs du ARM Cortex M4 ont créé un microprocesseur qui supporte deux systèmes indépendants de piles : 1-les piles des tâches 2-la pile pour les interruptions :



Le ARM Cortex M4 contient une banque de pointeurs de pile pour économiser de la mémoire RAM avec un système d'exploitation préemptif: le MSP est SP par défaut. Cependant, il est possible de configurer le système pour que PSP soit SP dans le main, c'est-à-dire, lorsqu'on ne traite pas une interruption.

Lorsque PSP et MSP sont utilisés, PSP est SP dans le main. À chaque fois qu'on empile/dépile une valeur ou qu'on réfère à R13 en dehors des interruptions, on réfère à PSP. Par contre, lorsqu'on fait des opérations sur la pile dans une interruption, on réfère à MSP.

Si une interruption survient à partir du main, l'adresse de retour, les drapeaux, LR, R0-R3 et R12 sont sauvegardés sur la pile indiquée par PSP. Ensuite, toutes les opérations de pile effectuées dans l'interruption se font dans la pile indiquée par MSP (incluant l'empilement de données si une interruption plus prioritaire survient!).

Pour implémenter un OS moins « RAMivore » avec deux pointeurs de pile, il faut opérer quelques changements au système précédent :

- Il faut créer une pile pour les interruptions dans laquelle SP sera le MSP
- On peut diminuer la taille de la pile de chaque tâche (256 à 128 dans mon exemple)
- Il faut écrire le registre du CONTRÔLE du cœur pour utiliser deux pointeurs de pile, après avoir correctement initialisé les valeurs de PSP et MSP (MSP est SP par défaut).
- Lorsqu'on veut empiler/dépiler R4 à R11 dans l'interruption du système d'exploitation, il faut considérer que SP est MSP et que la pile de la tâche interrompue (tâche active) utilisait PSP.
- Il faut aussi considérer que le code « caché » du compilateur exécuté au début/à la fin de l'interruption utilise MSP : les piles des tâches ne contiendront plus EXC_RETURN ou les registres utilisés dans l'interruption.

Main	Taches	Interruption du SysTick

//char PileMain[**128**];
 	
void main(void)
{
__disable_interrupt();
initSystick();
initPileT1();
initPileT2();
initPileT3();
PileActive = &pPileMain;
initDeuxPointeursDePile();
__enable_interrupt();
 	
while(1)
{};
}
	char PileT1[**128**];
	char PileT2[**128**];
	char PileT3[**128**];
	
	void T***x***(void)
	{
	initT***x***();
	while(1)
	{
	ExecuteT***x***();
	}
	}
	
	Où ***x*** peut valoir 1, 2 ou 3…
		char** PileActive;
		char *pPileMain;
		char *pPileT1;
		char *pPileT2;
		char *pPileT3;
		char PileInt[128];
		
		void SysTick_Handler(void)
		{
		SauvegardeContexte();
		PileActive = ChoixTacheAExecuter();
		ChargementContexte();
		}

En assembleur, les fonctions modifiées deviennent :

SauvegardeContexte :

```
;The stack pointer of active process is in process stack pointer that must be retrieved
mrs    r2, PSP ;Get stack pointer of active process into r2

;All process information except R4 to R11 is saved upon interrupt entry
stmdb r2!, {r4-r11}      ;PUSH R4 to R11

;Save stack pointer of active process
ldr r0, = PileActive
ldr r1, [r0]  ;r1 contains the address of the active process information
str r2, [r1]  ;Store stack pointer of active process into TCB
BX    lr     ;Return
```

ChargementContexte:

```
;Get stack pointer of active process
ldr r0, = PileActive
ldr r1, [r0]  ;r1 contains the address of the active process information
ldr r2, [r1]  ;put stack pointer of active process into r2

ldmia r2!, {r4-r11}      ;POP R4 to R11
msr    PSP, r2 ;After OS interrupt, process will be restored from PSP stack
BX    lr     ;Return
```

initDeuxPointeursDePile: ;Note: Cette fonction corrompt r0, r1 et r2!!!

```
;Use PSP instead of MSP for main process and declare a new stack for interrupts using MSP
;Assume sp is MSP (default) and function is called in Thread mode (main)
mov r0, sp    ;Get and store location of current main stack
ldr r1, =PileInt    ;Get and store location of interrupt stack
add r1, r1, #SIZE_OF_PILE_INT       ;Interrupt stack is full descending
mov sp, r1    ;Store interrupt stack into MSP
mov r2, #2    ;Set CONTROL special register to use PSP in thread mode
msr CONTROL, r2       ;Set CONTROL special register to use PSP in thread mode
isb;     ;Flush pipeline
mov sp, r0    ;Restore main stack into PSP
BX    lr     ;Return
```

Erreurs possibles d'application

Le code ci-dessus et les exemples ci-dessus ne considèrent pas quelques erreurs qui peuvent se produire en pratique:

- Le OS devrait détecter des erreurs de débordement de pile, pour chaque processus.
    - Habituellement, le OS met une séquence précise d'octets à la fin de la pile de chaque processus : si cette séquence change, la pile a débordé.
- Avec un FPU et lorsque des fractions sont utilisées, il faut gérer les registres du FPU sur la pile de chaque processus.
    - Voir http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0298a/BCGHEEFD.html pour corriger efficacement.
- Le code ci-dessus assume que le compilateur n'écrira pas de registre à la sortie de l'interruption du OS (seuls R0, R1, R2, R3 et R12 sont utilisés dans cet interruption), après avoir effectué le chargement du nouveau contexte. Par exemple, si le compilateur fait un POP R4 à la fin de l'interruption du OS, ce registre sera corrompu.
    - Tout le code entre **SauvegardeContexte()** et **ChargementContexte();** devrait toujours être inclus dans une fonction unique.