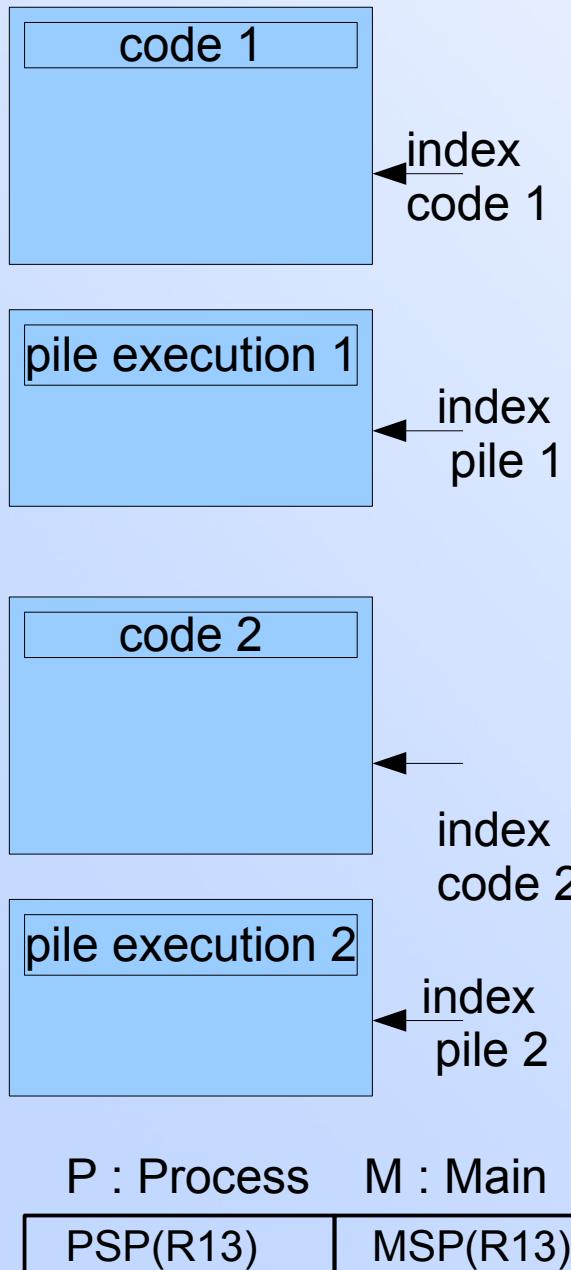


Ressources



IT changement de contexte

- sauvegarder le contexte de la tâche 1 sur la pile 1
Automatique, puis complété

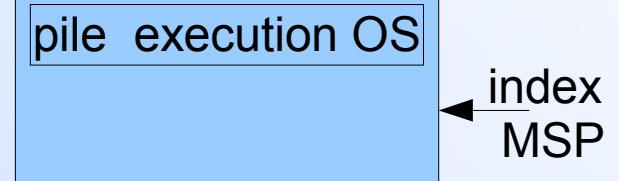
écrire l index de pile 1 dans le descripteur de tâche 1

- choisir la prochaine tâche :

= changer de descripteur

= CHANGER DE PILE PSP

- restituer le contexte de la tâche 2 depuis la pile 2 à la main, puis automatique au retour d'IT change contexte



La pile de chaque tâche, à la reprise contient le contexte complet :

- tous les registres dont
- APSR (status register)
- index du code

```
_asm void xPortPendSVHandler( void )
{
    // en venant dans cette exception LR14 contient 0xFFFFFFFF
    extern uxCriticalNesting;
    extern pxCurrentTCB;      // contient l'adresse du TCB de la tâche qu'on quitte
    extern vTaskSwitchContext;
    PRESERVE8
    // FIN DE SAUVEGARDE DU CONTEXTE DE LA TACHE QUE L'ON QUITTE
    mrs r0, psp    // instruction en mode handler permettant de récupérer SP(R13) Thread

    isb
    ldr r3, =pxCurrentTCB    // Récupérer l'adresse de la variable pointeur pxCurrentTCB

    ldr r2, [r3]    // R2 contient maintenant l'adresse du descripteur de tache courante (TCB)

    stmdb r0!, {r4-r11} // Sauver les registres restants non sauvés par le passage en IT
                        // sur la pile de la tâche en cours dont l'adresse est dans R0

    str r0, [r2]        // Sauver la position de la pile dans le TCB (c'est son premier élément).

    // Il faut maintenant faire décider quelle tâche exécuter ....
    // APPEL A L'ORDONNANCEUR DE TACHE pour déterminer la tâche suivante.
```

Commuter de tâche:
étape 2 : APPEL A L'ORDONNANCEUR DE TACHE
pour déterminer la tâche suivante.

// Dans la fonction vTaskSwitchContext, pxCurrentTCB sera mis à jour avec la tâche ciblée.

```
stmdb sp!, {r3, r14} // on protège R3 (adresse de la variable pxCurrentTCB)  
// et LR (R14) pour le retour d'interruption (valeur 0xFFFFFFFF)
```

```
mov r0, #configMAX_SYSCALL_INTERRUPT_PRIORITY // valeur 5<< 3  
msr basepri, r0 // mise à jour du registre basepri avec une instruction msr  
// uniquement utilisable en mode Handler ou thread privilégié  
// seules les ITs de priorité très élevées seront autorisées (valeur inférieure à 40 = 5<<3)  
// toutes ces interruptions n'ont pas le droit d'utiliser des appels système
```

```
dsb // Data Synchronization Barrier : attente de la fin des transferts en cours  
isb // Purge du pipeline d'instructions
```

```
bl vTaskSwitchContext //appel à la fonction vTaskSwitchContext() R14 est écrasé  
//pxCurrentTCB est mis à jour avec la tâche ciblée.
```

```
mov r0, #0  
msr basepri, r0 //toutes les ITs sont à nouveau autorisées (basepri =0;)  
ldmia sp!, {r3, r14} //récupération de R3 et R14
```

//pxCurrentTCB a été mis à jour avec la tâche ciblée.

```
ldr r1, [r3]          // récupération de l'adresse du TCB de la nouvelle tâche courante
ldr r0, [r1]          //récupération de l'adresse de la pile de la tâche courante dans son TCB
ldmia r0!, {r4-r11}   //récupération des registres non sauvés lors de la commutation
msr psp, r0           //mise à jour du SP (R13) du mode Thread
isb                  // Purge du pipeline d'instructions
bx r14               //Fin d'interruption (le processeur le sait par la valeur 0xFFFFFFF_ )
nop                 // pour le pipeline
}
} // fin de __asm void xPortPendSVHandler( void )
```

Comportement du compilateur conséquence sur le lancement de tâche

- Simuler un appel à la tâche comme si la première commutation de contexte avait lieu exactement pendant un faux appel de fonction
 - Lors d'un appel d'une fonction avec un paramètre , seul R0 est utilisé
Ce paramètre est un pointeur sur une structure sensée être connue
 - Si la tâche rend la main... ce n'est pas normal, toute tâche est comme une fonction main qui contient une boucle infinie de type while(1)
L'os devra alors reprendre la main :
On doit fabriquer un faux retour de fonction

2023 Appel de fonction: passage des paramètres

```

66: void init_var(unsigned char a, unsigned char b,\n
    unsigned char c, unsigned char d, unsigned char e)\n
0x00000260 B530      PUSH    {r4-r5,lr}\n
67: {uc_a=a;\n
0x00000262 9D03      LDR     r5,[sp,#0x0C]\n
0x00000264 4C1B      LDR     r4,[pc,#108]\n
0x00000266 7020      STRB    r0,[r4,#0x00]\n
68:     uc_b=b;\n
0x00000268 7061      STRB    r1,[r4,#0x01]\n
69:     uc_c=c;\n
0x0000026A 70A2      STRB    r2,[r4,#0x02]\n
70:     uc_d=d;\n
0x0000026C 70E3      STRB    r3,[r4,#0x03]\n
71:     uc_e=e;\n
0x0000026E 7125      STRB    r5,[r4,#0x04]\n
72: }\n
0x00000270 BD30      POP     {r4-r5,pc}\n
73: //----- Main Program -----\n\n
74: int main (void) {\n
0x00000272 B508      PUSH    {r3,lr}\n
75:     init_var(1,2,3,4,5);\n
0x00000274 2005      MOVS    r0,#0x05\n
0x00000276 9000      STR     r0,[sp,#0x00]\n
0x00000278 2304      MOVS    r3,#0x04\n
0x0000027A 2203      MOVS    r2,#0x03\n
0x0000027C 2102      MOVS    r1,#0x02\n
0x0000027E 2001      MOVS    r0,#0x01\n
0x00000280 F7FFFFEE  BL.W   init_var (0x00000260)\n
76:     init_proc();\n

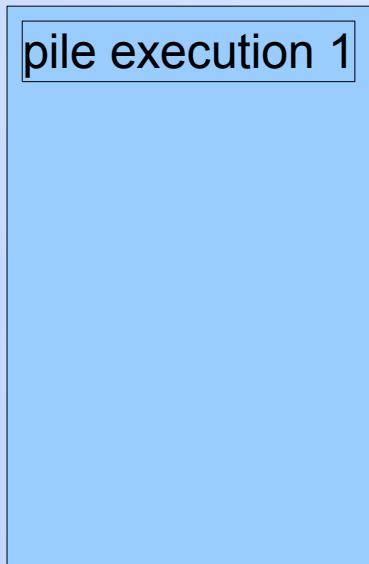
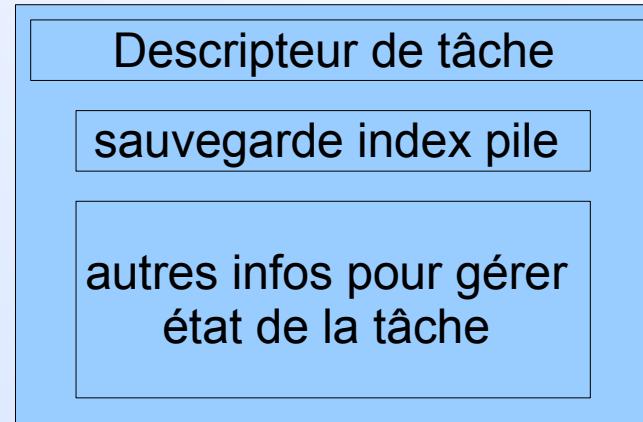
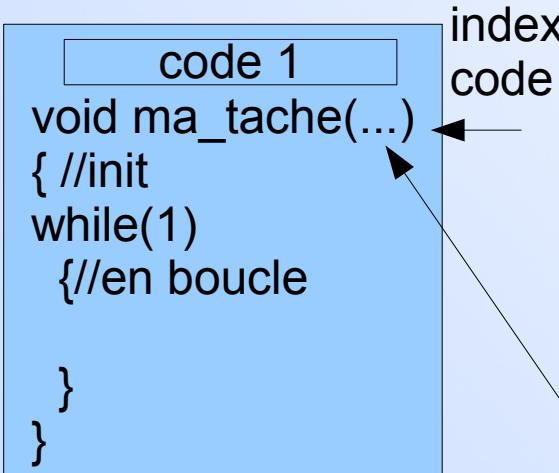
```

adresse	valeur
0x100001C4	
0x100001C8	
0x100001CC	
0x100001D0	
0x100001D4	
0x100001D8	
0x100001DC	
0x100001E0	
0x100001E4	
0x100001E8	
0x100001EC	
0x100001F0	
0x100001F4	
0x100001F8	
0x100001FC	
0x10000200	
<i>r₀</i>	
<i>r₁</i>	
<i>r₂</i>	
<i>r₃</i>	
<i>r₄</i>	
<i>r₅</i>	
<i>r₆</i>	
<i>r₇</i>	
<i>r₈</i>	
<i>r₉</i>	
<i>r₁₀</i>	
<i>r₁₁</i>	
<i>r₁₂</i>	
<i>r₁₃ (sp)</i>	
<i>r₁₄ (lr)</i>	
<i>r₁₅ (pc)</i>	

Quatre premiers paramètres
dans R0, R1, R2, R3
les suivants dans la pile

les Registres hors paramètres (R0 -R3) doivent être préservés pour le retour

Ressources



Paramètre à passer via le registre R0

La pile de chaque tâche, à la reprise contient le contexte complet :

- tous les registres dont
- APSR (status register)
- index du code

P : Process M : Main

PSP(R13)

MSP(R13)

IT changement de contexte classique pour le premier lancement de la tâche

– préparer une fausse pile :

choisir la valeur de chaque registre :

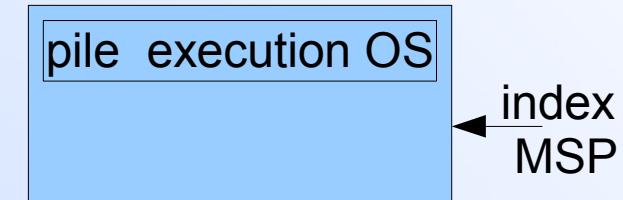
pour tous les registres dont l'état n'est pas important, on met leur numéro, pour voir si on récupère les registres dans le bon ordre :

R5 : 0x05050505

R12 : 0x12121212

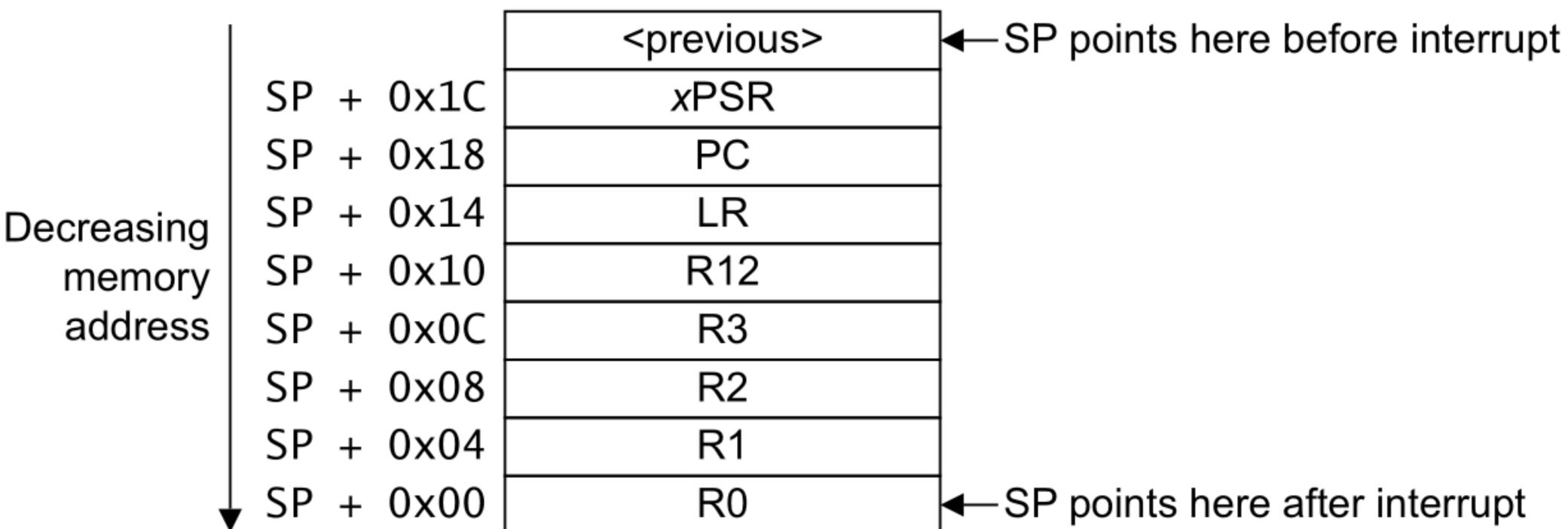
que faire pour :

PC, APSR, SP, LR, R0 ?



Sauvegarde de contexte

Pile lors d'une interruption



Il y a une sauvegarde automatique partielle du contexte :

A compléter d'une sauvegarde des registres R4 R5 R6 R7 R8 R9 R10 R11

A compléter d'une sauvegarde de ce lieu de stockage pour la future restitution de contexte quand la tâche reprendra la main

Initialisation de la pile pour première commutation

```
StackType_t *pxPortInitialiseStack( StackType_t *pxTopOfStack, \
                                    TaskFunction_t pxCode, void *pvParameters )

{pxTopOfStack--; // pile est de type FULL DESCENDING : décrémentation préalable
 *pxTopOfStack = portINITIAL_XPSR; // sauver un faux aPSR sur la pile

    pxTopOfStack--;
    *pxTopOfStack = ( ( StackType_t ) pxCode ) & portSTART_ADDRESS_MASK; // PC(R15)

    pxTopOfStack--;
    *pxTopOfStack = ( StackType_t ) prvTaskExitError ; // faux LR(R14) avec vrai retour

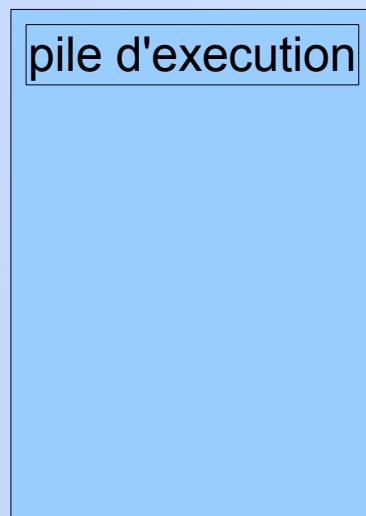
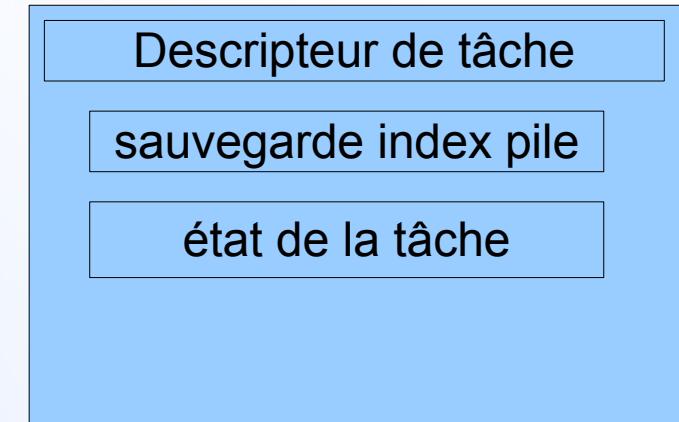
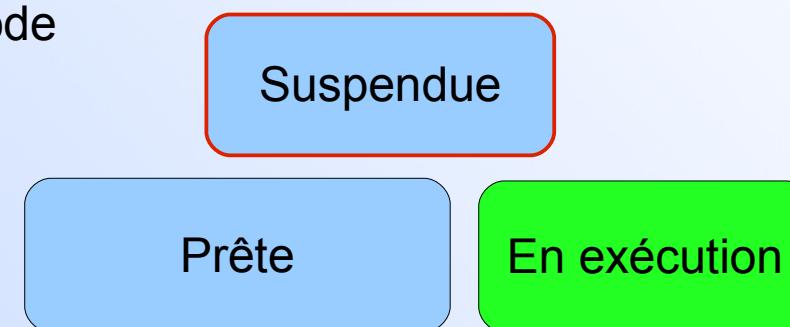
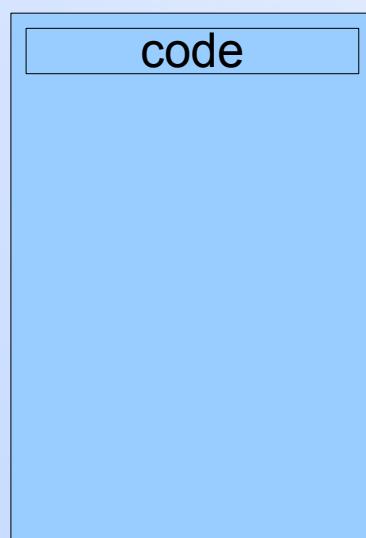
    pxTopOfStack -= 5; // la pile doit maintenant contenir R12, R3, R2, R1, R0
    // Seul R0 a besoin d'être initialisé, on simule un appel à une fonction, il contient
    // l'adresse de la structure de paramètres à passer à l'appel de cette fausse fonction
    // les tâches sont donc toutes déclarées avec la même structure pour être compatibles
    *pxTopOfStack = ( StackType_t ) pvParameters; // faux contenu de R0

    pxTopOfStack -= 8; // la pile doit maintenant contenir R11, R10, R9, R8, R7, R6, R5 et R4

    return pxTopOfStack; //pour la ranger dans le TCB
}
```

Que faut-il comme concepts à minima pour décrire informatiquement une Tâche dans l'idée de multi-tâches ?

Ressources



en attente d'évènement :

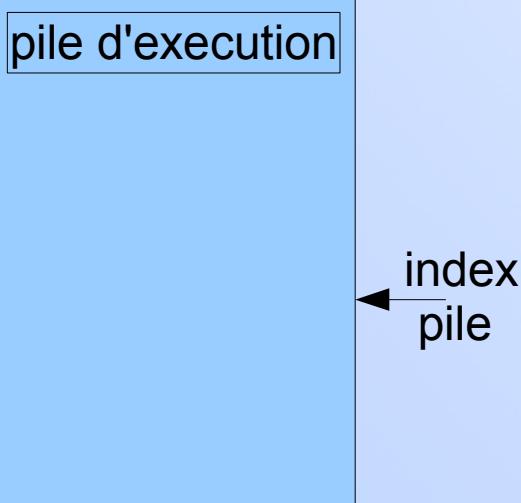
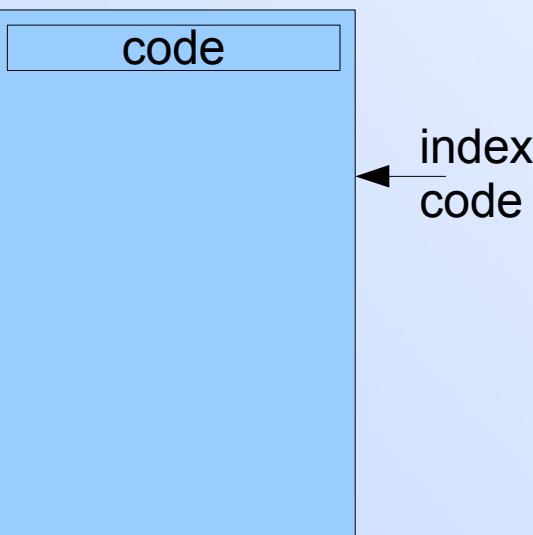
- attente délai
- évènement externe
- sémaphore ou signalisation venant d'une autre tâche

OS TEMPS REEL
ORDONNANCEUR

- gérer la liste des tâches
- savoir quelle tâche est active
- gérer le changement de tâche
- gérer le temps
- fournir un contexte de gestion évènementielle

Ressources :

1 pile par Tâche !
codes différents ou pas



1 TCB par Tâche (tableau)

status possibles :

- détruite
- suspendue
- attente
- execution :
 - priorité basse
 - priorité intermédiaire b
 - priorité intermédiaire h
 - priorité haute

Descripteur de tâche

sauvegarde index pile

Status de la tâche

Priorité si execution

Gestion delay attente

OS TEMPS REEL
ORDONNANCEUR

- gérer la liste des tâches
- savoir quelle tâche est active
- gérer le changement de tâche
- gérer le temps

Décider quelle est la tâche suivante en cas de commutation de tâche

// La fonction ci dessous est appelée lors de la seconde étape de la commutation
// dans l'interruption PendSV_Handler

```
void vTaskSwitchContext( void ) // à rendre le plus performant possible
{ // on doit rechercher dans la liste des tâches celle qui a la priorité la plus élevée
  // pour lui passer la main
  // on doit aussi en cas de tâches de priorités égales passer la main à la suivante
  // donc ce n'est pas une boucle For toute simple
```

```
// mettre à jour la variable pxCurrentTCB qui en assembleur gère la commutation
pxCurrentTCB = &OS_LISTE_TCB[tcb_choisi];
}
```

Cette technique a pour défaut de ne pas être à temps constant

Gestion du temps : Mise en attente / réveil

```
void Task_Delay( UInt32_t val_delay )
{ // calculer l'heure de réveil, la mettre dans le TCB

    // selon si le délai est infini ou pas, changer le status en suspended ou attente

    // provoquer une commutation de tâche car quelqu'un d'autre doit prendre la main

}
```

```
void SysTick_Handler( void ) // à rendre le plus performant possible
{ // Incrémenter le Tick , l'heure vient de changer

    // Parmi toutes les tâches en attente, y en a t il une à réveiller car c'est l'heure de son réveil?

    // Si besoin déclencher une commutation de tâche :
        // -- en cas de tâche réveillée plus prioritaire
        // -- en cas de plusieurs tâche de même niveau pour assurer le partage du CPU

}
```

Cette technique a pour défaut de ne pas être à temps constant

- SVC_Handler
- Pend_SV_Handler
- SysTick_Handler

il faut écrire la fonction qui permet de créer la tâche :

- écrire le prototype d'une fonction TASK_Create qui prend comme paramètres :
 - l'adresse du code de la tâche
 - l'adresse d'une structure permettant de passer des paramètres
 - les informations liées au concept de pile propre à la tâche :
 - taille de la pile souhaitée
 - emplacement de la pile (allocation non dynamique de mémoire)
 - la priorité de la tâche souhaitée
 - les informations lié au concept de TCB
 - adresse du TCB si allocation statique de memoire
 - pointeur vers un TCB pour récupérer éventuellement l'adresse du TCB si allocation dynamique de mémoire
- écrire le code de TASK_Create :
 - configurer la pile de la tâche pour permettre la première commutation vers la tâche
 - configurer le TCB associé à la tâche :
 - rendre la tâche connue de l'OS : inscription à des listes

il faut écrire la fonction qui permet de lancer l'OS :

- créer la tâche IDLE, tâche qui prend la main quand aucune tâche est éveillée
- initialiser les variables propres à l'OS :
 - variable qui compte les Ticks
 - variable qui contient l'adresse du TCB de la tâche en cours
- paramétrier et activer les interruptions propres à l'OS
- lancer la première tâche :
 - à la différence d'un changement de tâche :
 - on a pas de sauvegarde de contexte à faire
 - il faut qu'au retour d'interruption la tâche qui prend la main soit en mode utilisateur et non en mode superviseur...