MI11

Systèmes temps réel

Eléments de conception
pour un nœud

Traitement séquentiel

- Acquisition
- Traitement
- Restitution des résultats

Prise en compte et traitement d'événement dans un délai donné …

Traitement d'événements en un temps borné et reproductible

- Nécessité de mise en œuvre de mécanismes particuliers
  - Parallélisme
  - Communication
  - Synchronisation
  - Contraintes temporelles
- Différentes approches :
  - Synchrone : capter périodiquement la dynamique du procédé
  - Réactif : répondre instantanément aux « stimulations » en provenance du procédé
  - Asynchrone : exécutif multitâches temps réel; prise en compte des tâches périodiques et apériodiques

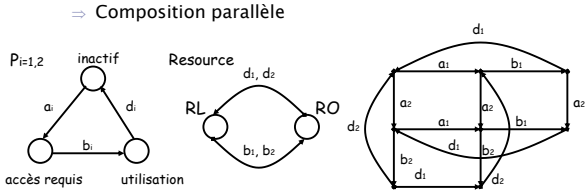
Approches synchrones – systèmes réactifs

- Hypothèse de synchronisme – réponse à tout événement extérieur synchrone à son occurrence
- Automate à états finis, GRAFCET, …
- Parallélisme mis en évidence hors ligne lors de l'analyse du problème
- Décomposition fonctionnelle + composition parallèle d'automate

Automate à états finis

$$M = \langle Q, \Sigma, \delta : Q \times \Sigma \to Q, q^0 \in Q \rangle$$

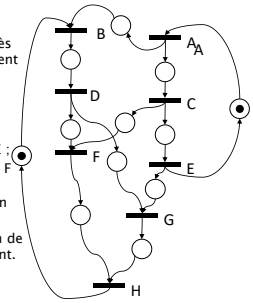
- Deux processus accèdent à une ressource en exclusion mutuelle
  - ⇒ Deux modèles indépendants
  - ⇒ Composition parallèle



Outils de spécification ⇒ programmation

- **Réseaux de Pétri – ordinaires, temporisés, synchronisés, …**
- Ordonnancement de 8 tâches non-interruptibles, A, B, C, D, E, F, G, H

- A l'initialisation, la tâche A est exécutable.
- Les tâches B et C ne peuvent être exécutées qu'après la fin A, ce qui n'implique pas que ces 2 tâches soient nécessairement lancées en même temps.
- La tâche D n'est exécutable qu'après la fin de B ;
- E n'est exécutable qu'après la fin de C ; F n'est exécutable qu'après la fin des tâches C et D ;
- G n'est exécutable qu'après la fin des tâches D et E ;
- Enfin, H n'est exécutable qu'après la fin des tâches F et G.
- La tâche A peut être à nouveau exécutée après la fin de E ;
- la tâche B peut être à nouveau exécutée après la fin de A et de H, et le cycle peut recommencer indéfiniment.



Interaction avec le « monde physique »

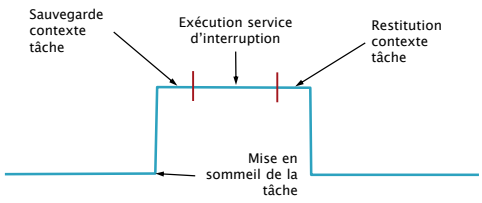
- **Interaction par scrutation**

```
Faire
  faire
      vérifier capteurs
  tant que données non disponibles
Lire capteurs
Traiter les données
Démarrer las actions
  faire
      vérifier les actionneur
  tant que actions effectuées
Jusqu'à arrêt du système
```

- **Avantages** : temps de réaction = temps d'une boucle, très simple
- **Inconvénients** : rigide, peu performant, gestion de plusieurs périphériques avec des fréquences de traitements différents

Interaction par interruptions

- **Interruption** : arrivé d'un événement qui interrompt l'exécution normale d'un programme



Limites de ces approches

- Grande partie du temps processeur consacrée à autre chose que l'application :
  - attente de ressources, de donnée,
  - terminaisons d'opérations en cours, …

⇒ Définition de mécanismes qui libèrent le processeur

⇒ Traitements immédiats et /ou différées de tâches

⇒ Systèmes multitâches

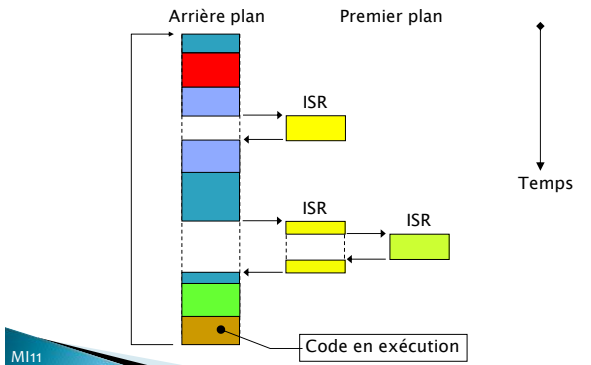
Multitâches Temps Réel

Structure d'un moniteur

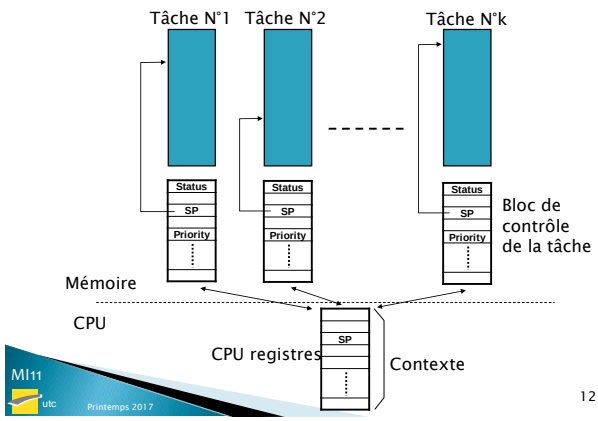
Notion de processus et tâches

Ressources

Organisation séquentielle



Division en processus



Processus

- **Définition** : Un processus est un ensemble d'actions qui se suivent naturellement les unes après les autres et qui sont indépendantes des autres actions n'appartenant pas au processus.
  - Conceptuellement chaque tâche possède un processeur virtuel comprenant sont pointeur d'instruction, sa zone de données, son pointeur de pile (le vecteur d'état).
  - En réalité, le processeur physique commute de tâches en tâches sous le contrôle de l'ordonnanceur.

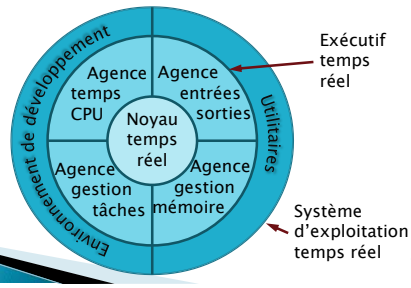
Processus = vecteur d'état + tâche

Tâche

- Deux parties : *programme (code) et données*
  - Le programme contient les instructions et éventuellement certaines constantes (coefficients, texte,…) – accessible uniquement en lecture.
  - La zone de données contient les données variables et les données constantes ou initialisées – accessible en lecture et en écriture.
- Vecteur d'état d'un processus
  - le contenu du compteur programme
  - le contenu des registres du microprocesseur
  - l'espace d'adressage du microprocesseur associé au processus
  - l'état des périphériques rattachés au microprocesseur

Exécutif temps réel

- gère les tâches d'une application de façon optimale
- offre des services de communication et de synchronisation des tâches.

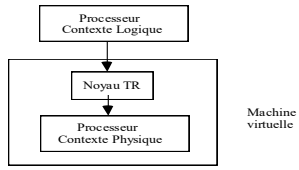


Noyau Temps Réel et Primitives

- NTR : gestion des différentes ressources du systèmes
  - simplifier la tâche du programmeur : fonctions entrées-sorties, gestion de la mémoire ...
  - services sous la forme de primitives
- Primitives : des séquences programmées grâce auxquelles l'utilisateur peut demander au moniteur l'exécution de fonctions déterminées
  - gestion des tâches
  - soulage le programmeur de certaines contraintes, telles que la gestion des ressources et des entrées/sorties.

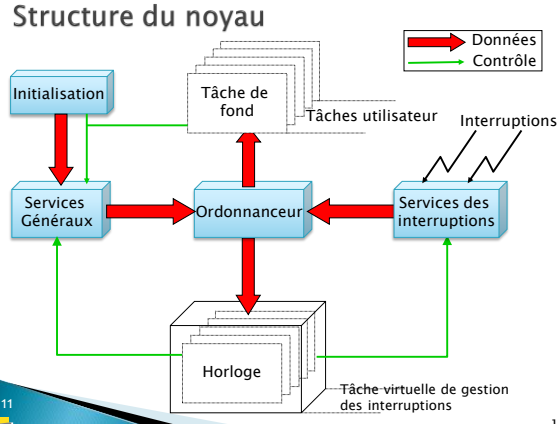
Noyau temps réel

- Evolution des processus sur un processeur physique.
- Ensemble de services situés au-dessus du jeu d'instructions du processeur central (physique) : un jeu de macro-instruction qui est le jeu d'instruction de la machine virtuelle.

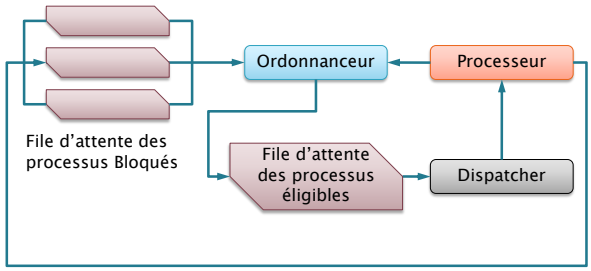


- partage du temps d'occupation du processeur
- protection des ressources communes
- gestion des interruptions
- gestion de la communication interprocessus
- primitives de gestion des processus

Structure du noyau



Sélection de processus

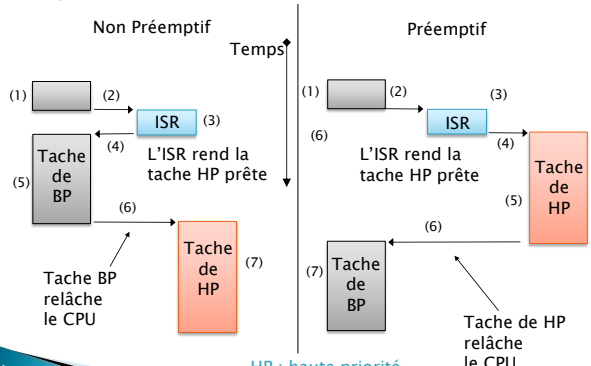


Les accès au noyau Temps Réel

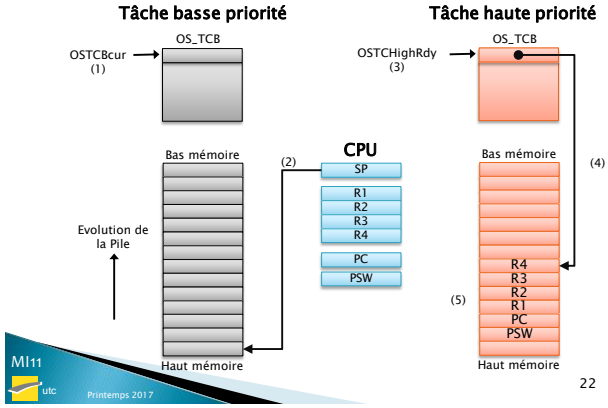
- **tâches matérielles**
- **tâches logiciel**

- **Un accès au noyau temps réel provoque** :
  - la sauvegarde du contexte de la tâche interrompue
  - l'exécution du programme d'interruption ou du service demandé
  - la détermination de la tâche prête la plus prioritaire
  - la restitution du contexte de cette tâche

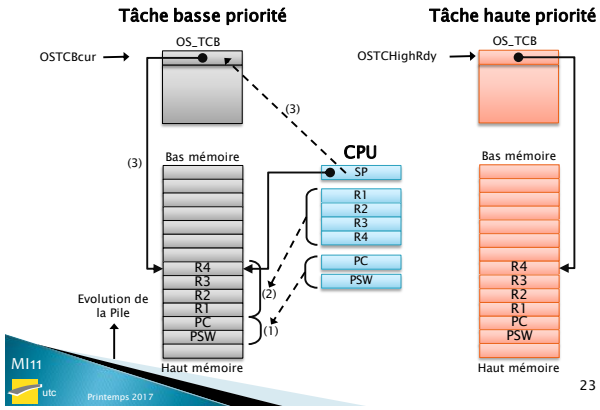
Noyaux



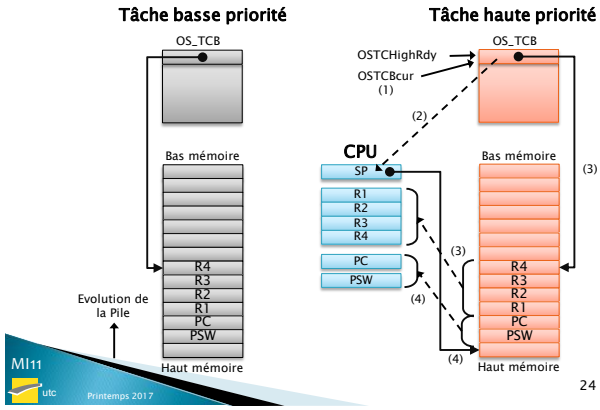
Commutation de contexte (1)



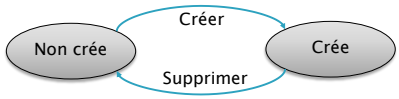
Commutation de contexte (2)



Commutation de contexte (3)

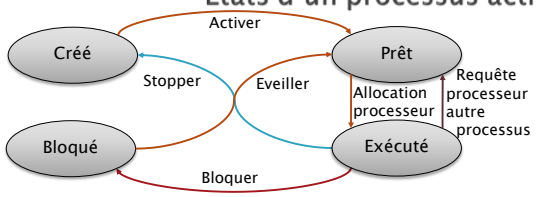


Etats des tâches



- **NON_CREE** :
  - la tâche est inconnu de l'exécutif
  - le code de la tâche est en mémoire
  - Les zones de données dynamiques et la pile ne sont pas attribués.
- **CREE** :
  - l'exécutif reconnaît la tâche.
  - La tâche devient processus (vecteur d'état, identificateur)
  - Les transitions : états NON_CREE ou PRET

Etats d'un processus actif



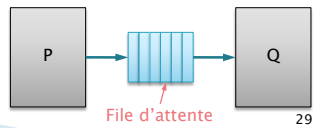
- Les franchissements des transitions entre les états sont la conséquence d'appels aux primitives du noyau.
- exclusivement appelées par le noyau (proposition du processeur à une tâche, attribution du processeur à une tâche, préemption d'une tâche),
- des services offerts à l'utilisateur (suspension, reprise, création et destruction d'une tâche).

Types de tâches

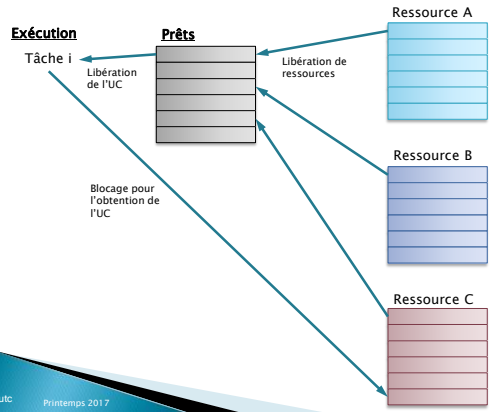
- **Périodiques** : elles ont des échéances strictes
- **Apériodiques** (sporadiques) : le début de l'exécution de la tâche est irrégulier (asynchrone)
- **Souple** : échéance relative (mieux tard que jamais)
- **Non temps réel** : aucune contrainte de temps

Ressources du processus

- Ressources : procédures, données, mémoire, processeur, fichiers, périphériques
- Ressources communes (partagées entre les différents processus)
  ⇒ L'évolution des processus sera donc dépendant de la disponibilité de la ressource:
  - **bloqué**
  - **actif**
- Types de ressources :
  - **ressource locale à un processus** : il lui appartient
  - **ressource commune** : partageable à n points d'accès, critique
- En fonction des ressources, les processus sont :
  - **indépendants**
  - **parallèles**
  - **en exclusion mutuelle**



Commutation sur ressources

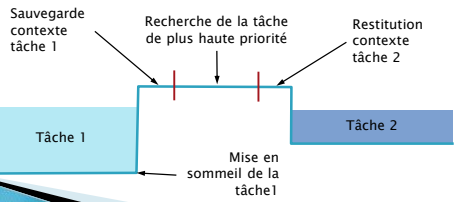


Contraintes temporelles

- Aucune contrainte temporelle
- Contraintes de temps essentiellement de trois types :
  - instant au plus tôt pour commencer une tâche
  - instant au plus tard pour la terminer
  - durée maximale d'exécution de la tâche
- Activation des tâches:
  - périodiquement (périodique)
  - à des instants fixes
  - de manière aléatoire (apériodique)

Interruption des tâches

- Plus de liberté à l'ordonnanceur
- Charge d'exécution induite par les changements de contextes
- Ressources en exclusion mutuelle, la préemption peut conduire à des situations d'interblocage.



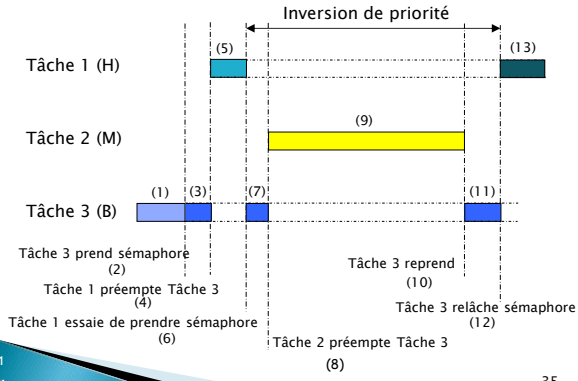
Relations entre tâches

- les tâches sont indépendantes : sans relation
- les tâches s'exécutent selon un ordre fixé
  ⇒ des contraintes de précédence
- les tâches partagent une (ou plusieurs) ressource(s) – contraintes de ressources
- Des algorithmes d'ordonnancement qui tiennent compte de :
  - Relations de précédence
  - Contraintes de ressources

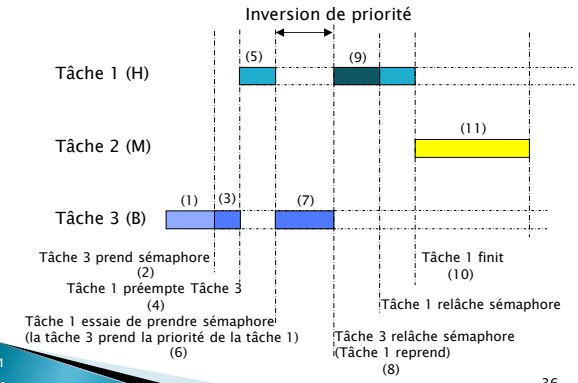
Priorités des tâches

- Critères pour l'allocation du processeur
- Priorités affectées aux tâches :
  - à la conception (priorités externes)
  - à partir d'un calcul fait par l'algorithme d'ordonnancement(priorités internes)
- Priorités statiques
- Priorités dynamiques

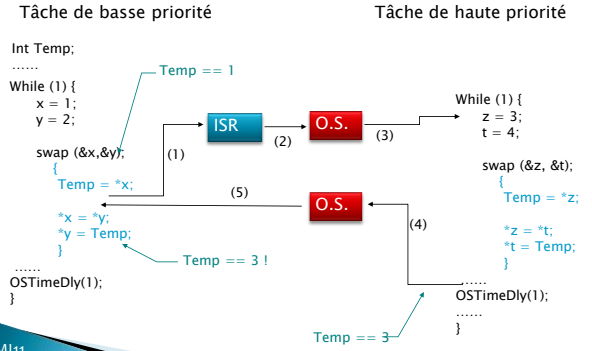
Inversion de priorité



Héritage de priorité



Non réentrance : ressource



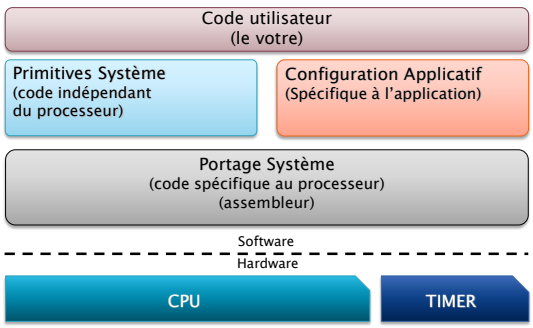
Quelques fonctionnalités

à travers l'exemple
du µC-OSII

Structure du noyau

Portabilité
Ordonnancement
Tâches

Dépendances processeur



Démarrage Noyau

- L'initialisation du noyau doit se faire avant tout appel à des éléments du noyau

```
OSInit();
```

- Une fois l'initialisation effectuée, des éléments peuvent être initialisés mais ne s'exécutent pas
- La prise en main du programme par le noyau se fait par l'appel d'une fonction spécifique :

```
OSStart();
```


Inhibition des interruptions

- Dans certain cas, il peut être important de désactiver les interruptions du système.
- Des primitives sont prévues à cet effet :
  - `OS_ENTER_CRITICAL();`
  - `OS_EXIT_CRITICAL();`

Bloc de contrôle de tâches

- Pour maintenir les informations concernant les tâches présentes dans le système, le noyau utilise des *Task Control Blocks*

```
typedef struct os_tcb {
    INT16U          OSTCBStkPtr;    /* Pointer to current top of stack */
#if OS_TASK_CREATE_EXT_EN > 0
    void            *OSTCBExtPtr;   /* Pointer to user definable data for TCB
                                       extension*/
    INT16U          OSTCBOpt;       /*Task options as passed by OSTaskCreateExt()*/
    INT16U          OSTCBId;        /* Task ID (0..65535) */
#endif
    INT16U          OSTCBStkBottom; /* Pointer to bottom of stack */
    INT32U          OSTCBStkSize;   /* Size of task stack */
    struct os_tcb *OSTCBNext;       /* Pointer to next     TCB in the TCB list */
    struct os_tcb *OSTCBPrev;       /* Pointer to previous TCB in the TCB list */
.....
    INT8U           OSTCBStat;      /* Task status */
    INT8U           OSTCBPrio;      /* Task priority (0 == highest, 63 == lowest)*/
.....
} OS_TCB;
```

Ordonnancement par priorité

- La tâche la plus prioritaire pouvant être exécutée est toujours mise en exécution

```
nodebug  void OS_Sched(void) {
    INT8U y;
    // don't use critical section macros since context switching inside
    // of a critical section can cause the counter to be incorrect
    OS_ENTER_CRITICAL();
    // Task scheduling must be enabled and not ISR level
    if ((OSLockNesting | bios_intnesting) == 0) {
        // Get pointer to highest priority task ready to run
        y             = OSUnMapTbl[OSRdyGrp];
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[ OSRdyTbl[ y ] ]);
        // No context switch if current task is highest ready
        if (OSPrioHighRdy != OSPrioCur) {
            OSTCBHighRdy = OSTCBPrioTbl[ OSPrioHighRdy ];
            OSCtxSwCtr++;              // Increment context switch counter
            OSCtxSw();
        }
    }
    OS_EXIT_CRITICAL();      // re-enable interrupts
}
```


Table de résolution de priorité

Index : modèle binaire pour résoudre la plus haute priorité
La valeur indexée correspond à la position du bit de plus haute priorité (0..7)

0x00 **INT8U OSUnMapTbl[ ]**

```
0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
```

OSRdyGrp = 01101000

OSRdyTbl[3] = 11100100

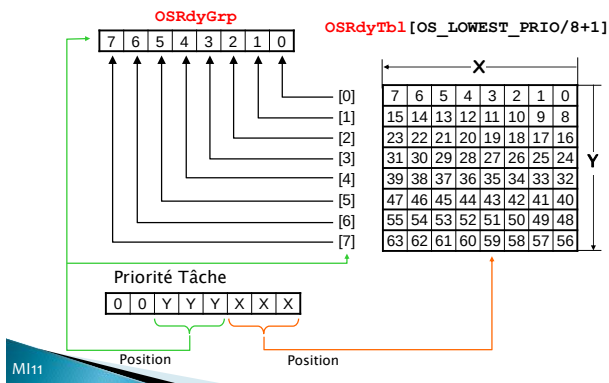
```
y = OSUnMapTbl[ OSRdyGrp ]

x = OSUnMapTbl[ OSRdyTbl[y] ]
```

Priorité Tâche

0	0	Y	Y	Y	X	X	X
0	0	0	1	1	0	1	0

0xFF

Liste des tâches prêtes



Rendre une tâche prête

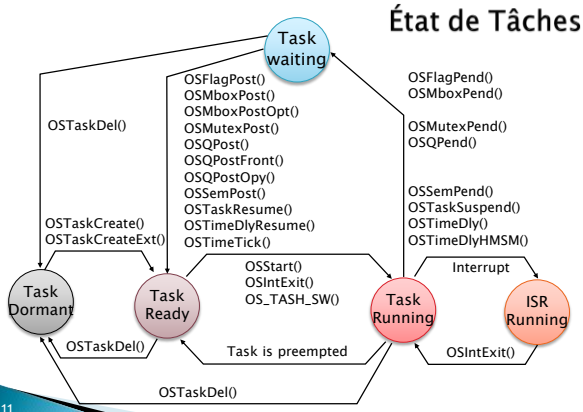
OSMapTbl[ ]

Index	Bit Mask (Binary)
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

```
OSRdyGrp               |= OSMapTbl[ prio >> 3 ]

OSRdyTbl[prio >> 3 ] |= OSMapTbl[ prio & 0x07 ]
```


Les Tâches

État de Tâches



Piles de tâches

- Chaque tâche se voit allouer une pile
- Des valeurs de piles sont définies par défaut ; cela peut être modifié dans le code

```
#define STACK_CNT_256   3 // number of 256 byte stacks
#define STACK_CNT_512   1 // number of 512 byte stacks
#define STACK_CNT_1K    2 // number of 1K stacks
#define STACK_CNT_2K    1 // number of 2K stacks
#define STACK_CNT_4K    0 // number of 4K stacks
```

- Des tâches cachées peuvent avoir besoin de piles (IDLE, application, Statistic)

Création de Tâches

```
INT8U OSTaskCreate
            (    void    (*task)(),
                 void    *pdata,
                 INT16U  stk_size,
                 INT8U   prio
            );
```

Pointeur sur l'adresse de la fonction de tâche

Pointeur sur des paramètres initiaux pour la tâche

Nombre d'octets dans la pile

Priorité, unique à chaque tâche

Return value

OS_NO_ERR : The call was successful

OS_PRIO_EXIT : Task priority already exists (each task MUST have a unique priority).

OS_PRIO_INVALID : The priority specified is higher than the maximum allowed (i.e. ≥ OS_LOWEST_PRIO)

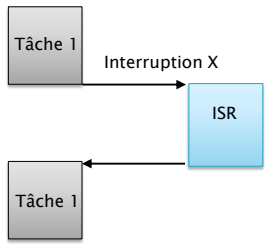
Format des Tâches

- Une tâche est :
  - une boucle infinie, qui doit contenir des primitive permettant de la mettre en pose
  - Une série d'instructions finissant par une primitive de fin de tâche

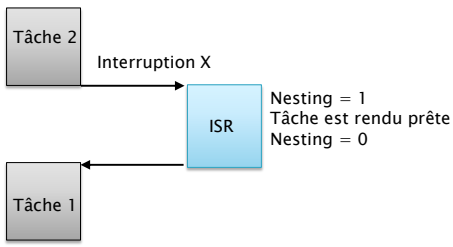
```
void Tache1(void *pdata)
    {
    short int n, flag;
    for ( ; ; )
        {
        OSSemPend(read_sem, 0, NULL);
        printf("Tache 1 \n");
        ….
        OSSemPost(FinRead_sem);
        }
    }
```

Les interruptions

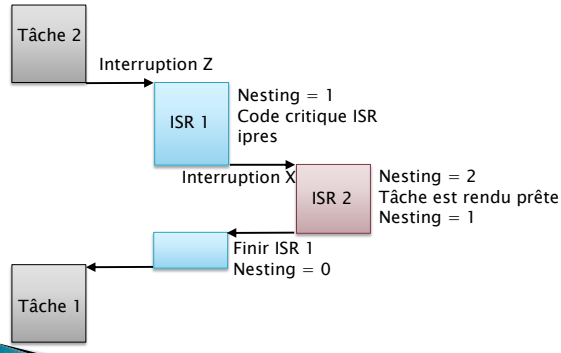
Interruptions : scénario de type 1



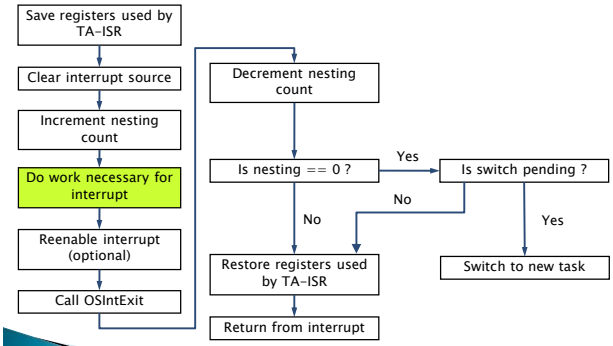
Interruptions : scénario de type 2



Interruptions : scénario de type 3



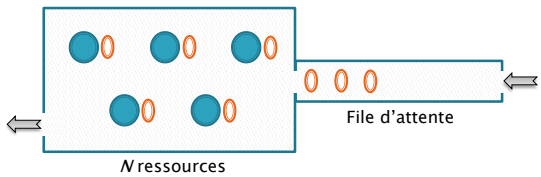
Interruptions : schéma global



Synchronisations
Sections critiques
Exclusions mutuelles

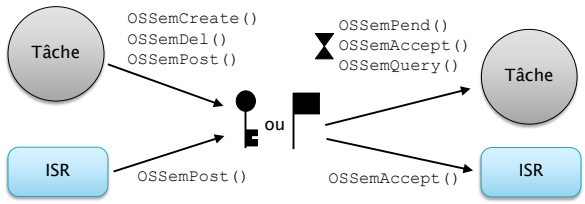
Sémaphore : définition

- Solution générale aux problèmes :
  - d'accès à des ressources partagées
  - de synchronisation inter-tâches



Sémaphore

- Relations entre tâches, ISR, sémaphore



Structure d'un Sémaphore

- Un sémaphore se présente sous la forme d'une structure de données initialisée par le noyau
- On accède au sémaphore par un pointeur sur la structure

```
.....
// Les sémaphores
OS_EVENT * user_sem;
OS_EVENT * read_sem;
.....
```

Création d'un sémaphore

```
OS_EVENT *OSSemCreate
         (
              INT16U    cnt
         );
```

Valeur initiale du sémaphore

```
.....
//On initialise les sémaphores
user_sem = OSSemCreate(0);
read_sem = OSSemCreate(0);
.....
```


Attente de jeton

```
void OSSemPend
         (    OS_EVENT  *pevent,
              INT16U    timeout,
              INT8U     *err
         );
```

Pointeur sur le block de contrôle du sémaphore désiré

Pointeur sur un message d'erreur

Temps en ticks d'attente de la ressource. Si 0, la tâche attends indéfiniment jusqu'à l'arrivée de l'événement

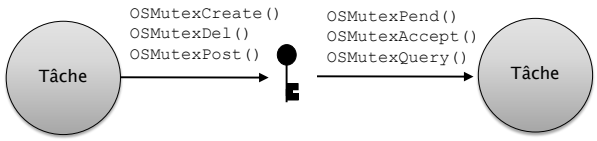
Envoi de jetons

```
INT8U OSSemPost
         (    OS_EVENT  *pevent
         );
```

Pointeur sur le block de contrôle du sémaphore désiré

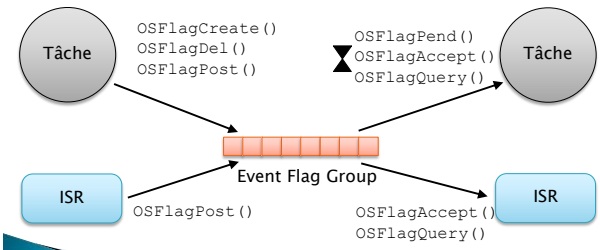
Mutex

- Version simplifiée du sémaphore



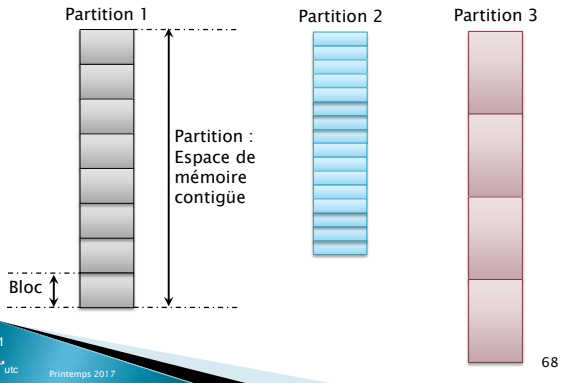
Gestion d'événements

- Attente simultanée sur une conjonction ou une disjonction d'événement



Gestion mémoire

Partition mémoire (1)



Partition mémoire (2)

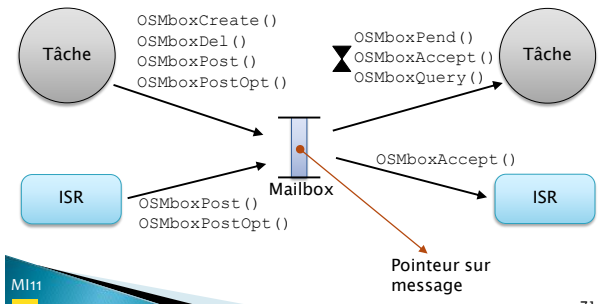
- Gestion par une liste chainée
- Mémoire allouée à l'initialisation
- Gestion de l'occupation des blocs par le noyau : allocation pseudo-dynamique
- Fourniture d'un pointeur vers le début du bloc

```
OSMemCreate()
OSMemGet()
OSMemPut()
OSMemQuery()
```


Messagerie Asynchrone

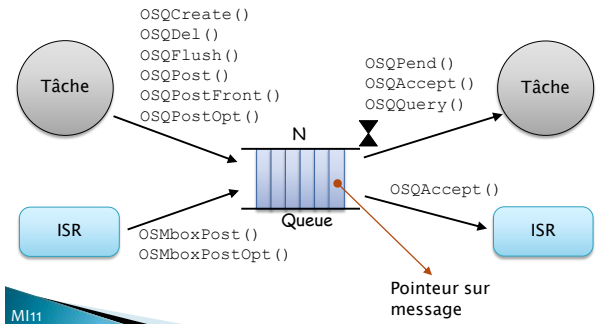
Mailbox

- Peut-être publique ou privée en fonction des noyaux

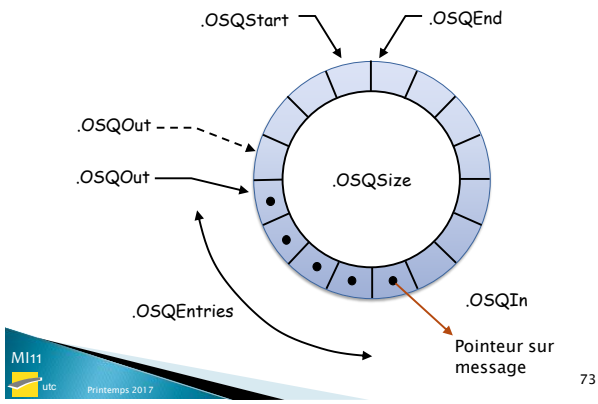


Queue de messages

- Relations entre tâches, ISR, et queue de messages



Queue : structure



Partition mémoire pour la messagerie