OS temps réel pour capteurs et actionneurs

B. Miramond
Polytech Nice Sophia

Les différentes solutions OS embarqué

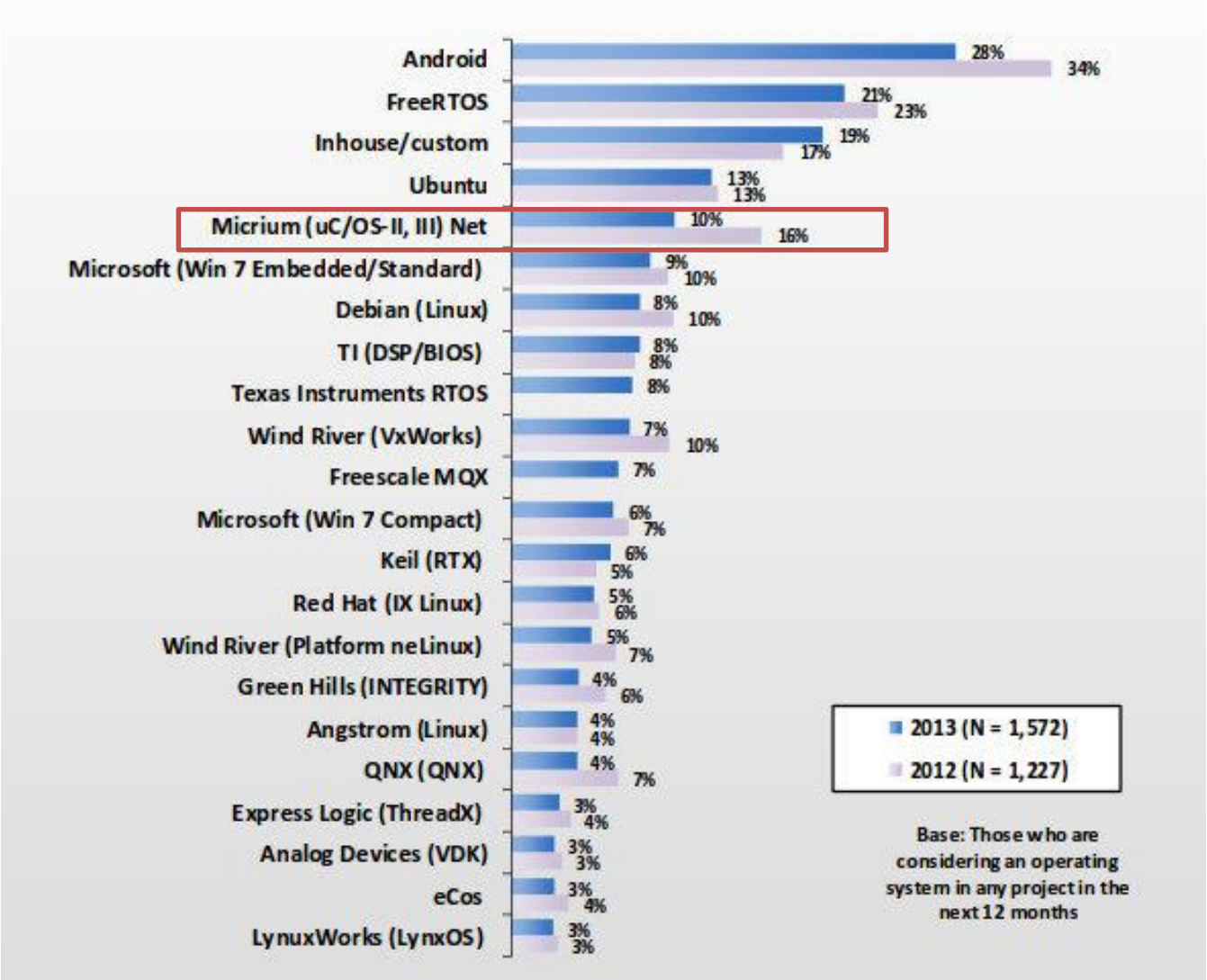
Vendor	OS	SMP
WindRiver	VxWorks 6.6 SMP	Y
eSol	eT-kernel Multicore Edition	Y
Express Logic	ThreadX	Y
QNX	Neutrino RTOS	Y
Green Hills	INTEGRITY 10	Y
Montavista	MobiLinux 5.0	Y
kernel.org	Linux 2.6+	Y
Symbian	Symbian OS 9+	Y
Microsoft	WinCE	(*)
Mentor Graphics	Nucleus PLUS RTOS	(*)
Solaris	Open Solaris	Y
etc	etc	(*)

(*) Contact vendor for further details
Vendors are at different stages of ARM MPCore support

Le choix d'un OS

- Le choix d'exécutif de type RTOS n'est pas forcément évident :
  - Taille mémoire supplémentaire (ROM & RAM)
  - Surcoût de 2 à 4 % de temps CPU
  - Coût d'achat de l'OS (de 70 à 30k $ ou royalties : 5 à 500 $ par unité)
  - Coût de maintenance (15% du coût de devt par an)
- A contrario : conception rapide, extensions faciles, pas de conception niveau OS…

Les solutions existantes



UBM 2013 Embedded Market Study

http://ubmcanon.com/design-and-manufacturing/2013EmbeddedStudy/index.html

Notre cas d'étude : noyau temps réel uC/OS-II

- uC/OS-II (Micro Controller Operating System)
  - 1992 (J. Labrosse)
  - 40 processeurs différents (8-64 bits)
  - Portable
  - ROMable
  - Scalable (modulaire)
  - Préemptif à priorités fixes
  - Déterministe
  - Pas de MMU

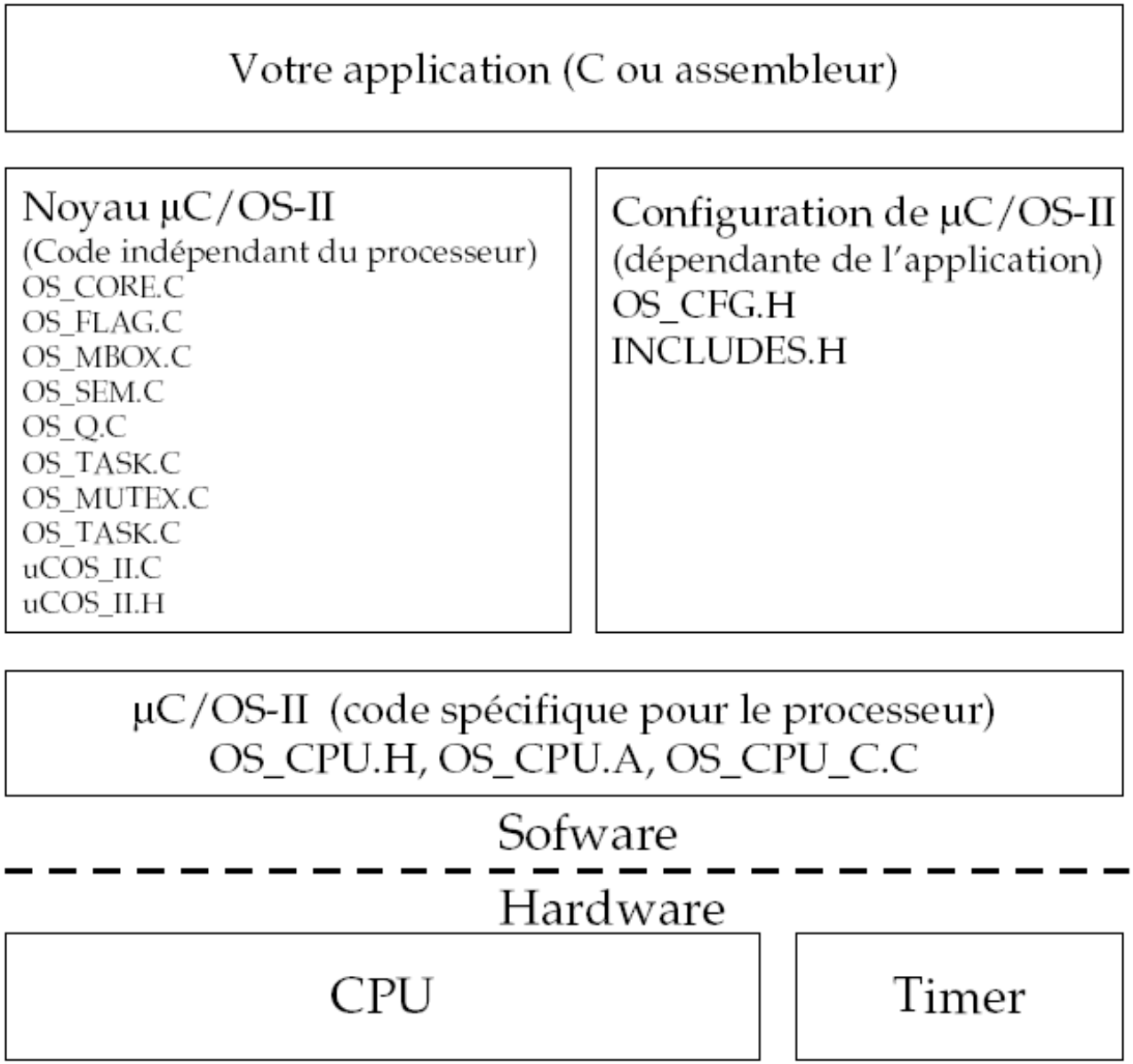
Tâches ou processus sous uC/OS-II

- 64 taches maximum
- Priorités exclusives
- Donc pas de round robin (2 tâches n’ont pas la même priorité)
- Les tâches peuvent communiquer grâce aux ISR (Interrupt Service Routine)

Services du noyau

- Le noyau uC/OS ne fournit que les services :
  - Ordonnancement
  - Changement de contexte
  - Gestion des modes de syncrhonisation (sémaphores, mutex)
  - Gestion des moyens de communication (mailbox, queue)
  - Retards et timout
- Ce qui donne une empreinte mémoire de quelques Ko.

Structure de uC/OS-II



Contrainte de taille mémoire

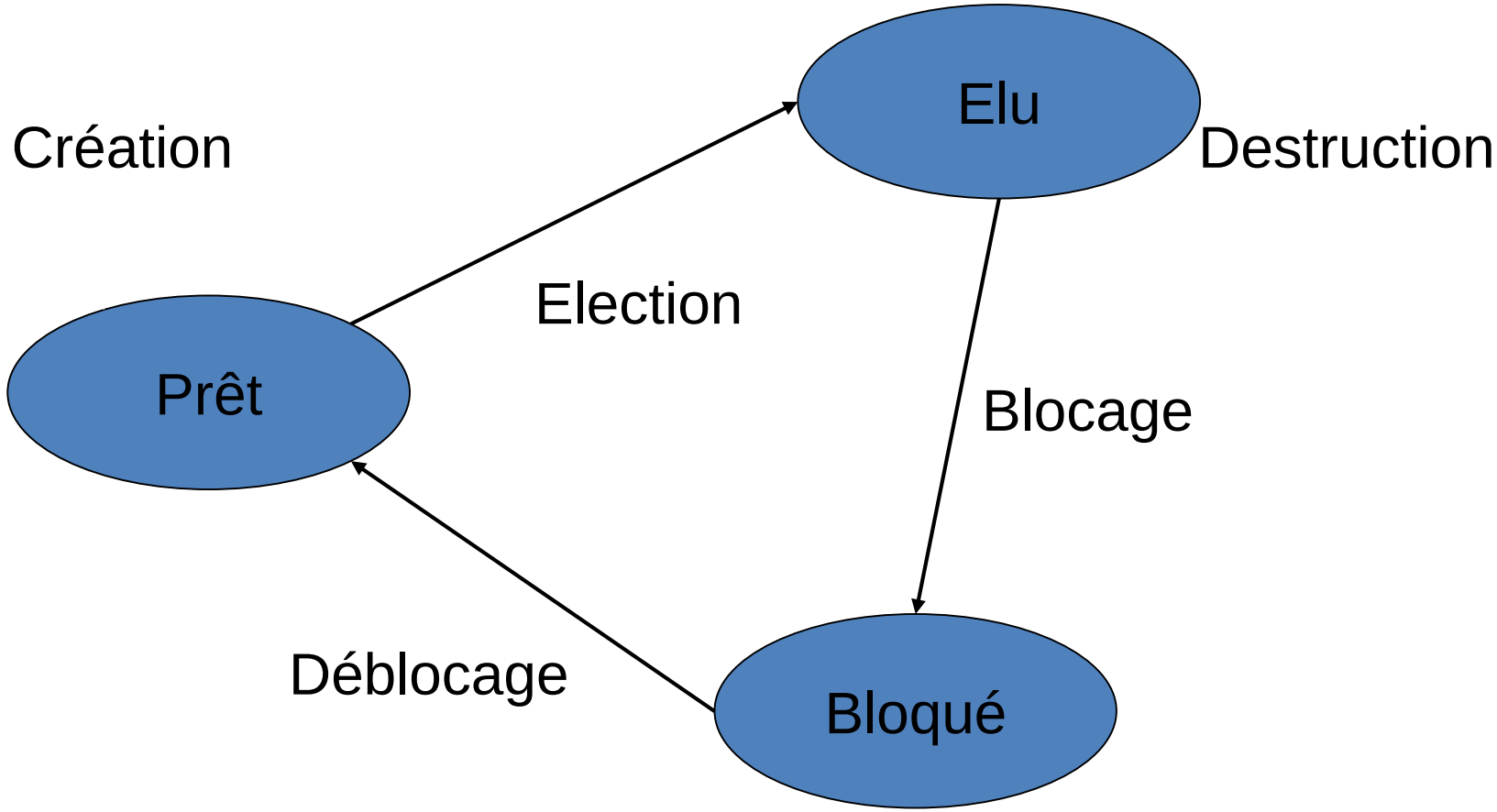
- La taille totale de l'application avec un RTOS :
  - Taille du code applicatif
  - Espace de données (RAM) et d'instructions (ROM) utilisées par le noyau
  - SUM(Task Stack)
  - MAX(ISR Nesting)

- Pour des systèmes avec peu de mémoire, il faut faire attention à l'évolution de la taille de pile :
  - Tableau et structures locaux aux fonctions
  - Appels de fonctions imbriqués
  - Nombre d'arguments des fonctions

Les tâches du système

- L'ordonnanceur de uC/OS-II gère les tâches suivantes :
- Les tâches utilisateur
- Plus 2 tâches système :
  - Idle Task (OS_LOWEST_PRIO)
  - Statistics Task (OS_LOWEST_PRIO - 1)
- Les priorités étant codées comme un INT8U, et que les priorités ne sont pas partagées (pas de round robin), le nombre de tâches possibles est de 64 – 2 = 62 tâches utilisateur.

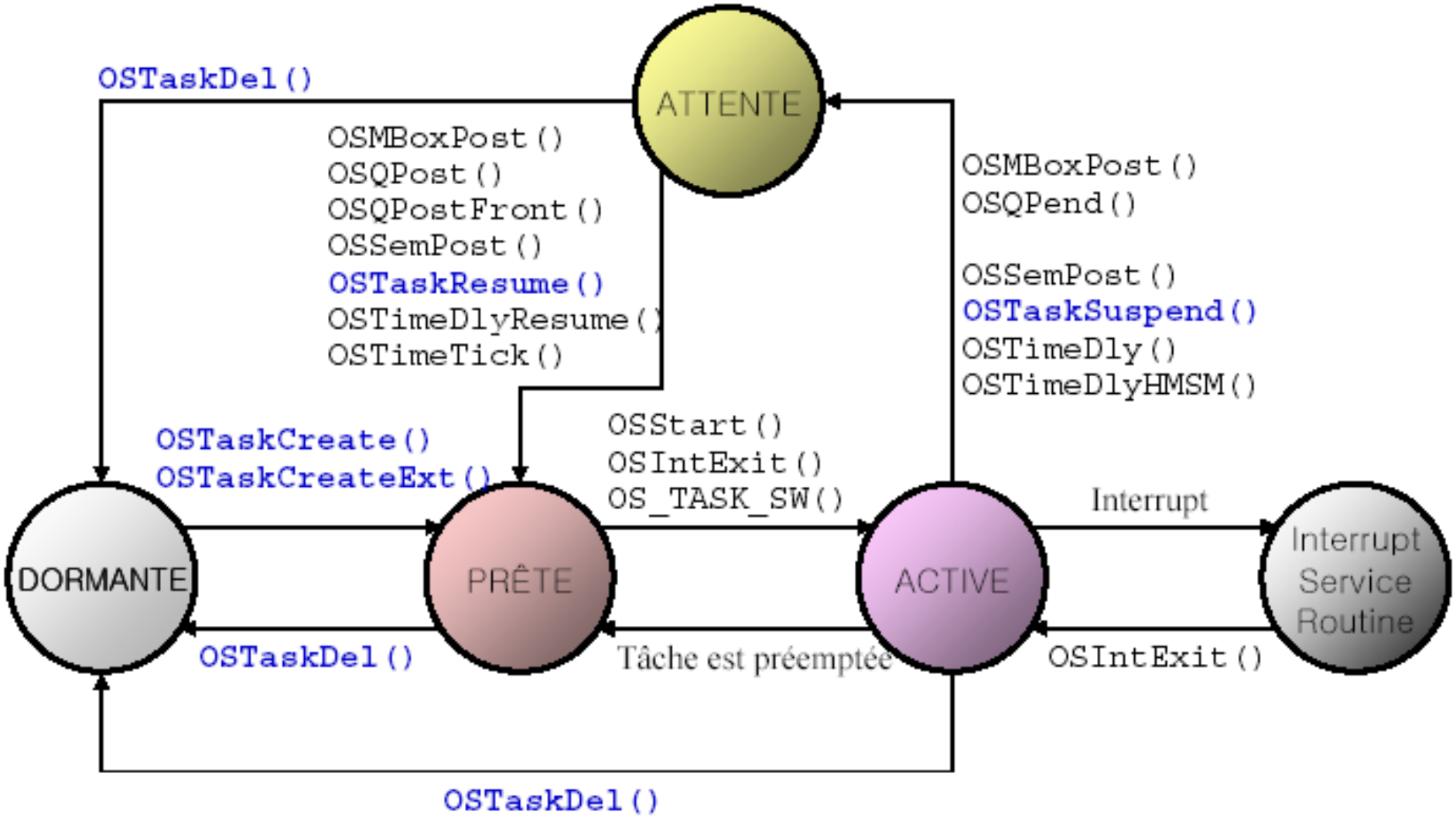
Diagramme d’état d’un processus



Commentaire sur l'état des tâches

- DORMANT :
  - Destruction de tâches (OSTaskDel())
- WAITING :
  - Attente événement ou timeout (OSTimeDly() ou OS_X_Pend())
- ISR :
  - Sur interruptions
  - Au retour la tâche courante peut-être préemptée par une plus prioritaire (passage en état RDY)
- READY :
  - Création de tâches avant le lancement du multitâches (OSStart())
  - préemptée

Diagramme d'état des processus dans uC/OS-II



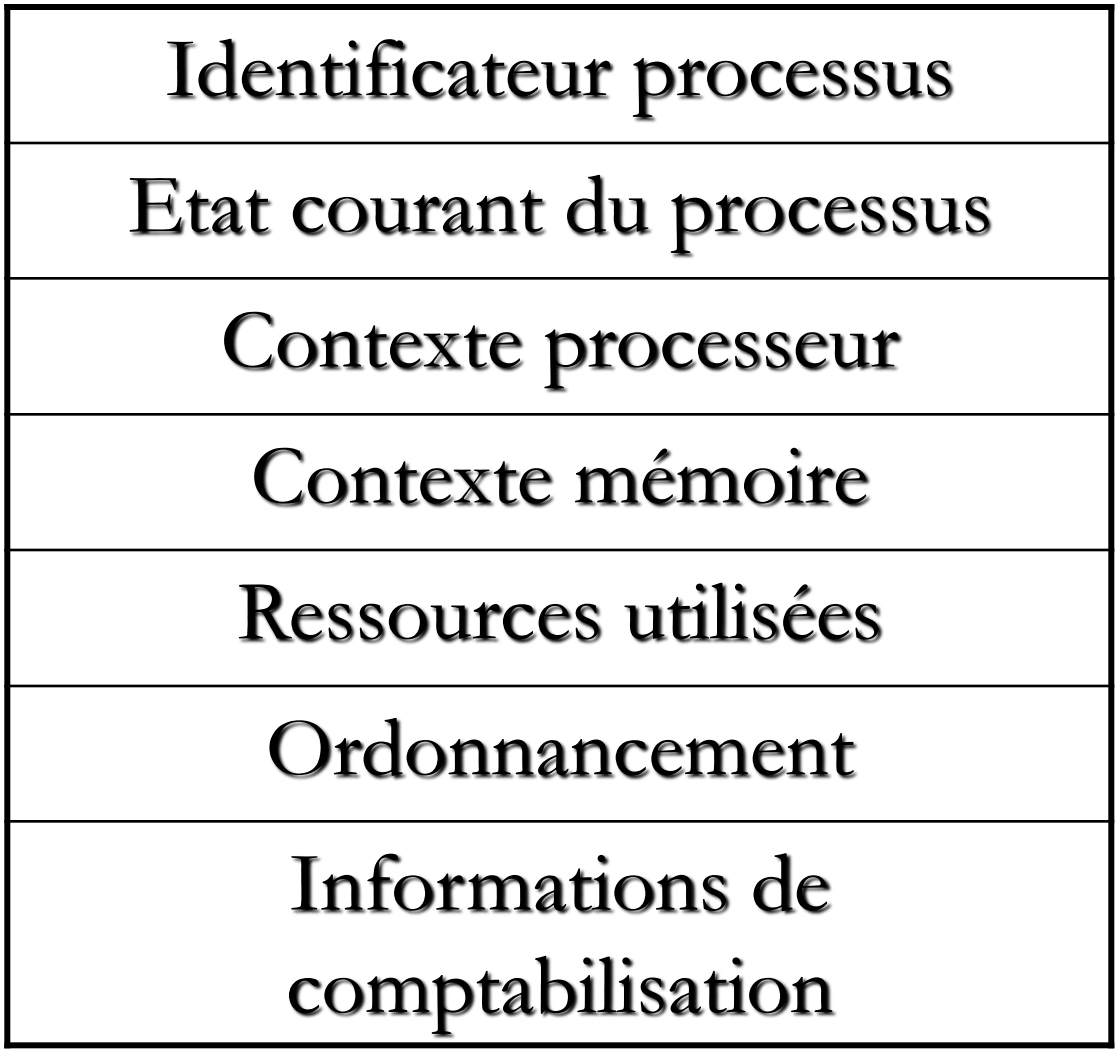
CONTEXTE D’UNE TACHE

Bloc de contrôle d'un processus

L'OS conserve le contexte d'une tâche dans une structure de données spécifique appelée le **TCB**.

Il y a changement de **contexte** (save/restore) à chaque pré-emption ou changement de tâche :

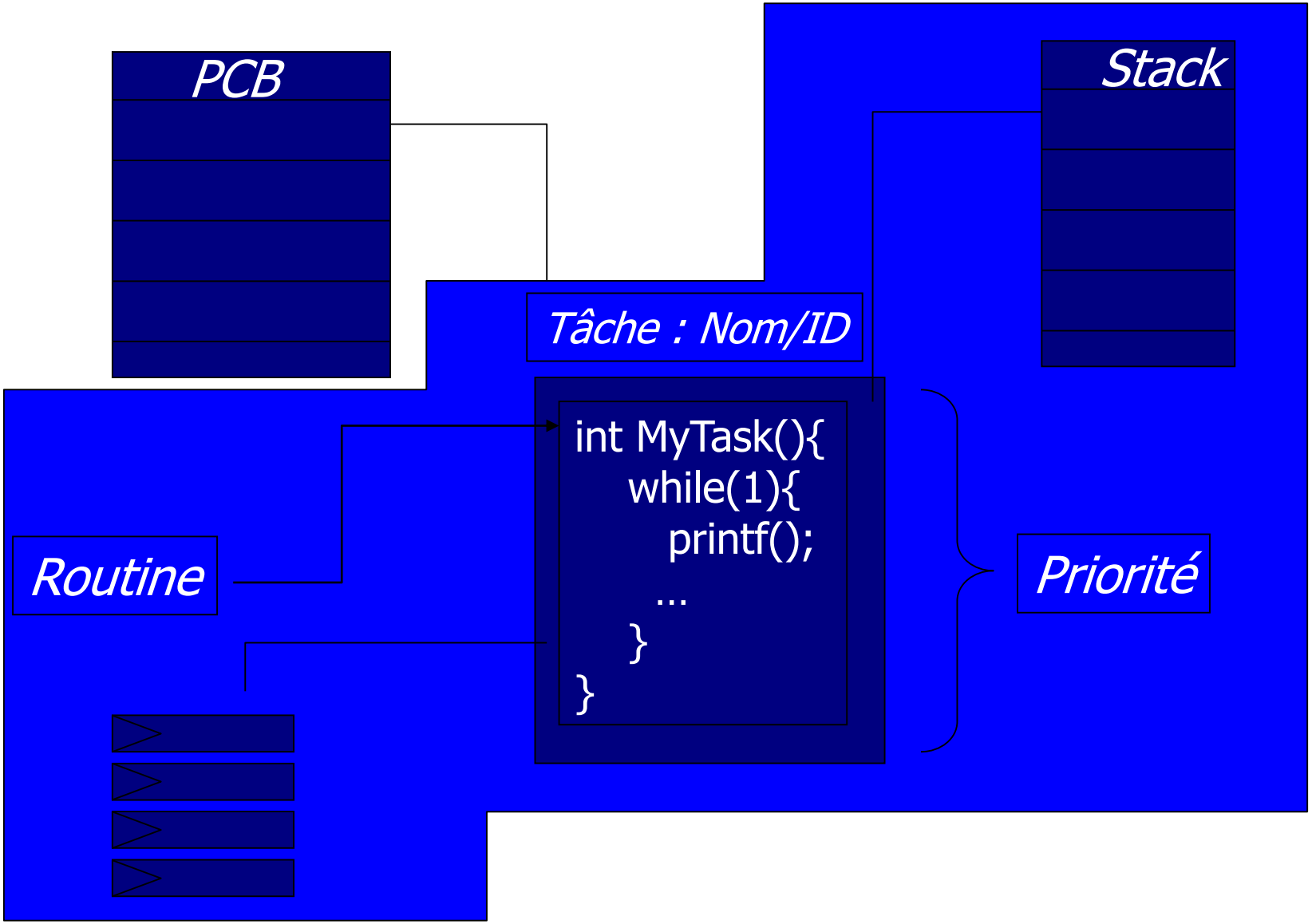
- Registres du processeurs
  - Registres généraux
  - PC, SP, registres de sections
  - Pile d'exécution
  - État de la tâche



Plus précisément

- L'identificateur du processus tel qu'il lui a été affecté à sa création (un entier)
- L'un des états du processus
- La valeur des registres du processeur (PC, RI, SR…)
- Les adresses de début et de fin de la pile d'exécution
- Les fichiers ouverts, les outils de synchronisation utilisés
- La priorité du processus, sa file d'attente…
- Le temps CPU utilisé, la taille de sa pile, …

Résumé sur le processus



Primitives (fonctions) de gestion des tâches sous µC/OSII

- ***OSTaskCreate( )*** : *Création d'une nouvelle tâche avec une priorité, un numéro d'identification, une pile…..*
- ***OSTaskDel( )***: Destruction de tâche
- ***OSTaskChangePrio( )*** : Modification de la priorité d'une tâche
- ***OSTaskSuspend ( )*** : Suspendre l'exécution d'une tâche
- ***OSTaskResume ( )*** : Rendre active une tâche précédemment suspendue
- ***OSTaskQuery( )*** : Retourne la priorité et l'état d'une tâche

Les fonctions de création de tâches : OSTaskCreate() et OSTaskCreateExt()

- Règle 1 : Les tâches doivent être créées avant le lancement de la fonction OSStart() qui lance le mécanisme de gestion multi-tâches.

- Règle 2 : Ext est une version étendue utilisée uniquement pour l'analyse de code.

  La première a 4 arguments, la seconde 9.

Arguments de la fonction de création

- void (*task)(void*pd): un pointeur sur le code de la tâche
- Void * pdata : un pointeur sur une donnée passée en paramètre à la tâche à sa création
- OS_STK *ptos : pointeur sur le haut de pile de la tâche
- INT8U prio : La priorité désirée de la tâche (limitations à 64 possibilités)
- INT16U id : Un identifieur unique de la tâche (extension par rapport à la limitation 64, sinon id = prio)
- OS_STK *pbos : Bottom-of-stack
- INT32U stk_size : taille de la pile (utilisée pour le stack checking). Cf. TP_Exercice 2.
- void *pext : pointeur sur une donnée utilisateur pour étendre le TCB. Cf. TP_Exercice3.
- INT16U opt : uCOS_ii.h contient la liste des options possibles (OS_TASK_OPT_STK_CHK, OS_TASK_OPT_TSK_CLR, OS_TASK_OPT_SAVE_FP…). Chaque constante est un FLAG binaire.

Fichier µCOS_II.h : TCB (1)

OS_STK = INT16U => affichage *2 en octets

Voir exemple d'utilisation en TP

Utiliser pour la mesure dynamique de taille de pile

Non utilisé : ID = prio

```
*********************************************************************************************************
*********************************************************************************************************
*/

typedef st              b {
    OS_STK          *OSTCBStkPtr;          /* Pointer to current top of stack                          */

#if OS_TAS      EATE_EXT_EN > 0
    void            *OSTCBExtPtr;          /* Pointer to user definable data for TCB extension         */
    OS_STK          *OSTCBStkBottom;       /* Pointer to bottom of stack                               */
    INT32U           OSTCBStkSize;         /* Size of task stack (in number of stack elements)         */
    INT16U           OSTCBOpt;             /* Task options as passed by OSTaskCreateExt()              */
    INT16U           OSTCBId;              /* Task ID (0..65535)                                       */
#endif

    struct os_tcb *OSTCBNext;              /* Pointer to next     TCB in the TCB list                  */
    struct os_tcb *OSTCBPrev;              /* Pointer to previous TCB in the TCB list                  */

#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0) || (OS_SEM_EN > 0) || (OS_MUTEX_EN > 0)
    OS_EVENT        *OSTCBEventPtr;        /* Pointer to event control block                           */
#endif
```

Fichier µCOS_II.h : TCB (2)

```
#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    void           *OSTCBMsg;            /* Message received from OSMboxPost() or OSQPost()          */
#endif

#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) &&
#if OS_TASK_DEL_EN > 0
    OS_FLAG_NODE   *OSTCBFlagNode;       /* Pointer to event flag node                               */
#endif
    OS_FLAGS        OSTCBFlagsRdy;       /* Event flags that made task ready to run                  */
#endif

    INT16U          OSTCBDly;            /* Nbr ticks to delay task or, timeout waiting for event    */
    INT8U           OSTCBStat;           /* Task status                                              */
    INT8U           OSTCBPrio;           /* Task priority (0 == highest, 63 == lowest)               */

    INT8U           OSTCBX;              /* Bit position in group  corresponding to task priority (0..7) */
    INT8U           OSTCBY;              /* Index into ready table corresponding to task priority    */
    INT8U           OSTCBBitX;           /* Bit mask to access bit position in ready table           */
    INT8U           OSTCBBitY;           /* Bit mask to access bit position in ready group           */

#if OS_TASK_DEL_EN > 0
    BOOLEAN         OSTCBDelReq;         /* Indicates whether a task needs to delete itself          */
#endif
} OS_TCB;
```

Utilisé si l'option OSTCBOpt.OS_TASK_EN = 1

wait(timeout) ou wait(event, timeout)

Évite les calculs en-ligne, cf. chapitre suivant

OSTaskDel

- OSTaskDel(OS_PRIO_SELF);
- Auto Destruction d’une tâche lorsqu’elle a terminé son traitement de manière à éviter d’occuper de la mémoire inutilement.

EXÉCUTION ET ORDONNANCEMENT DE TÂCHES SUR UN RTOS

```
void VotreTache(void *pdata)
{
    Faire quelque chose avec l'argument' pdata;
    Initialisations;
    for (;;) {
        /* execution (votre Code)                          */
        attente d'un évenement;/* un délai ...             */
                               /* Signal d'une ISR ... */
                               /* Signal d'une tâche...*/
        /* execution (votre Code)                          */
    }
}
```

Endless-loop () {

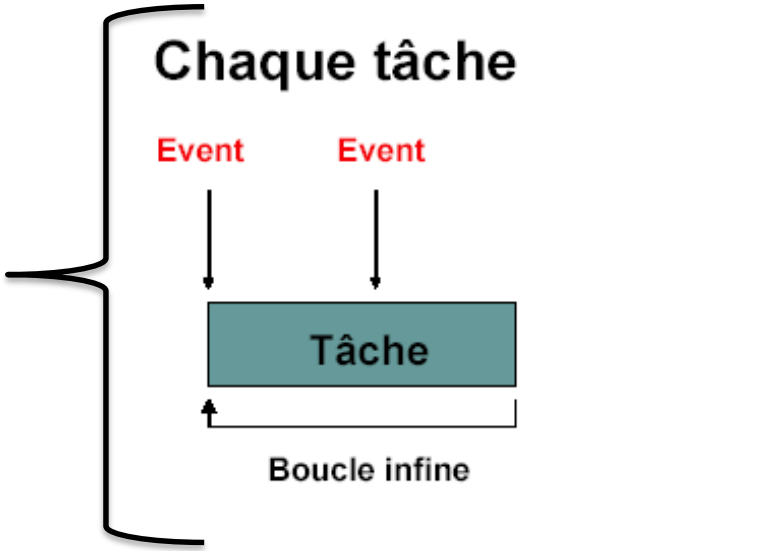
***Boucle infinie** {*

Corps de boucle

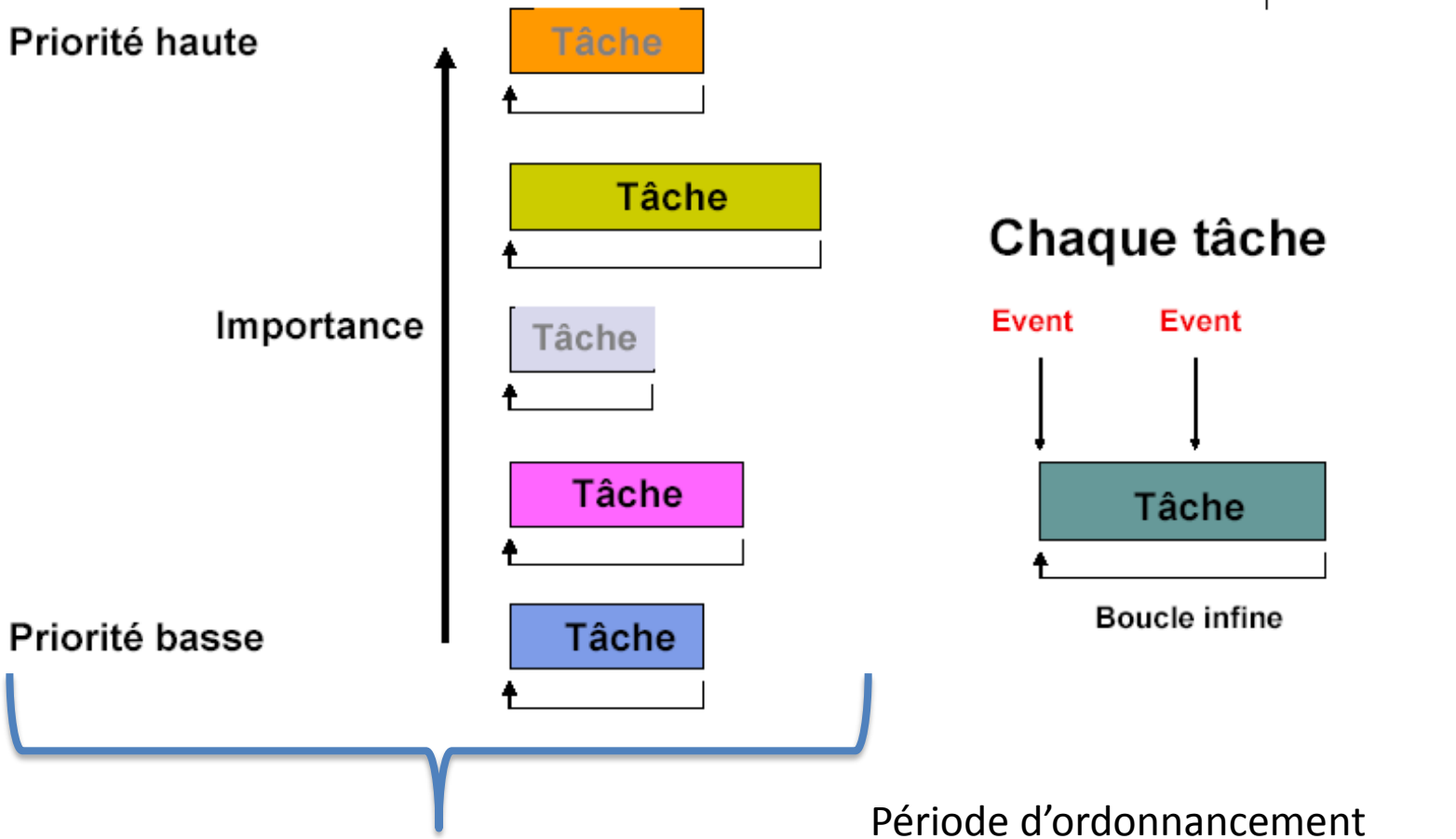
Appels bloquants

}

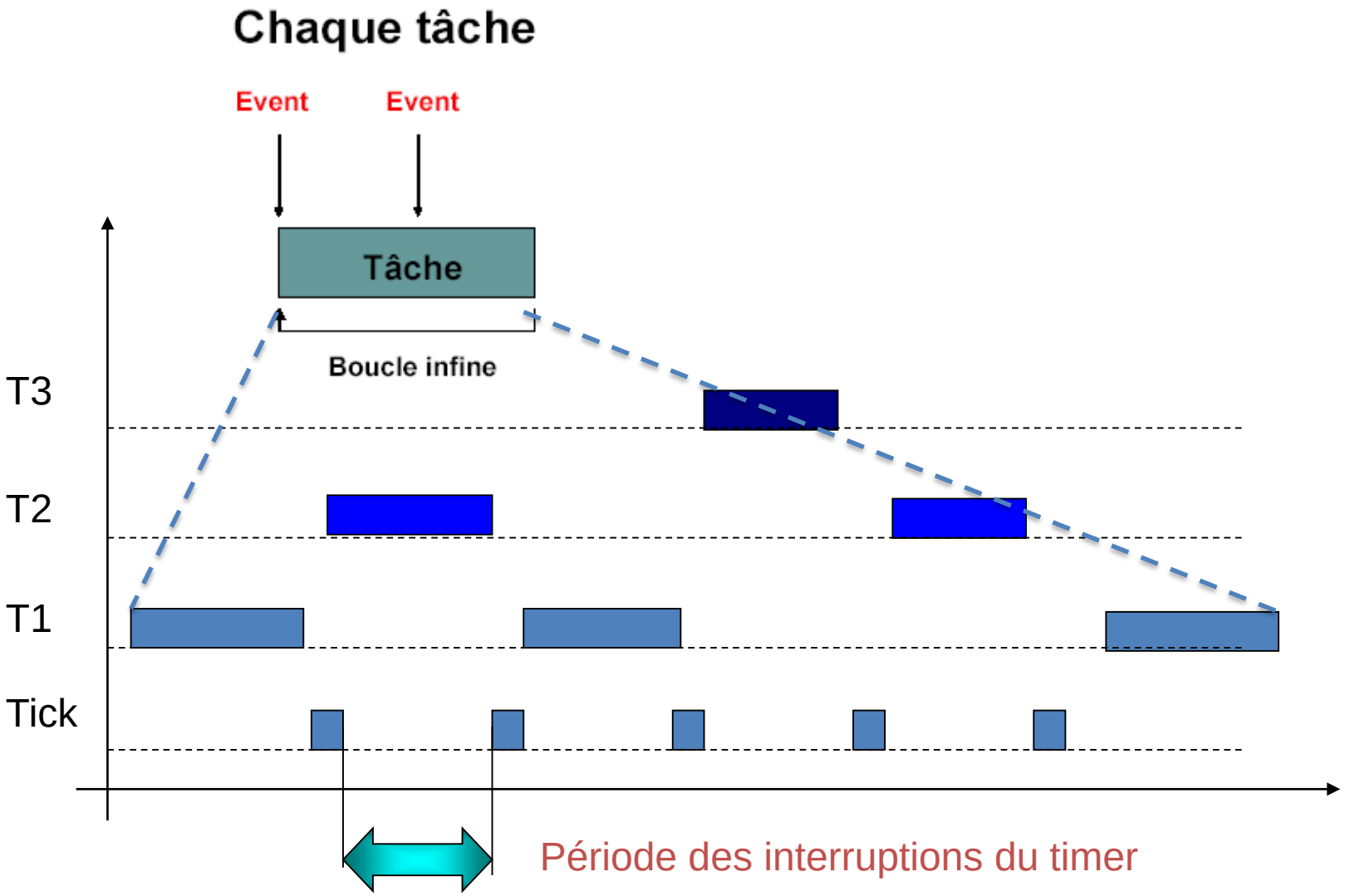
}



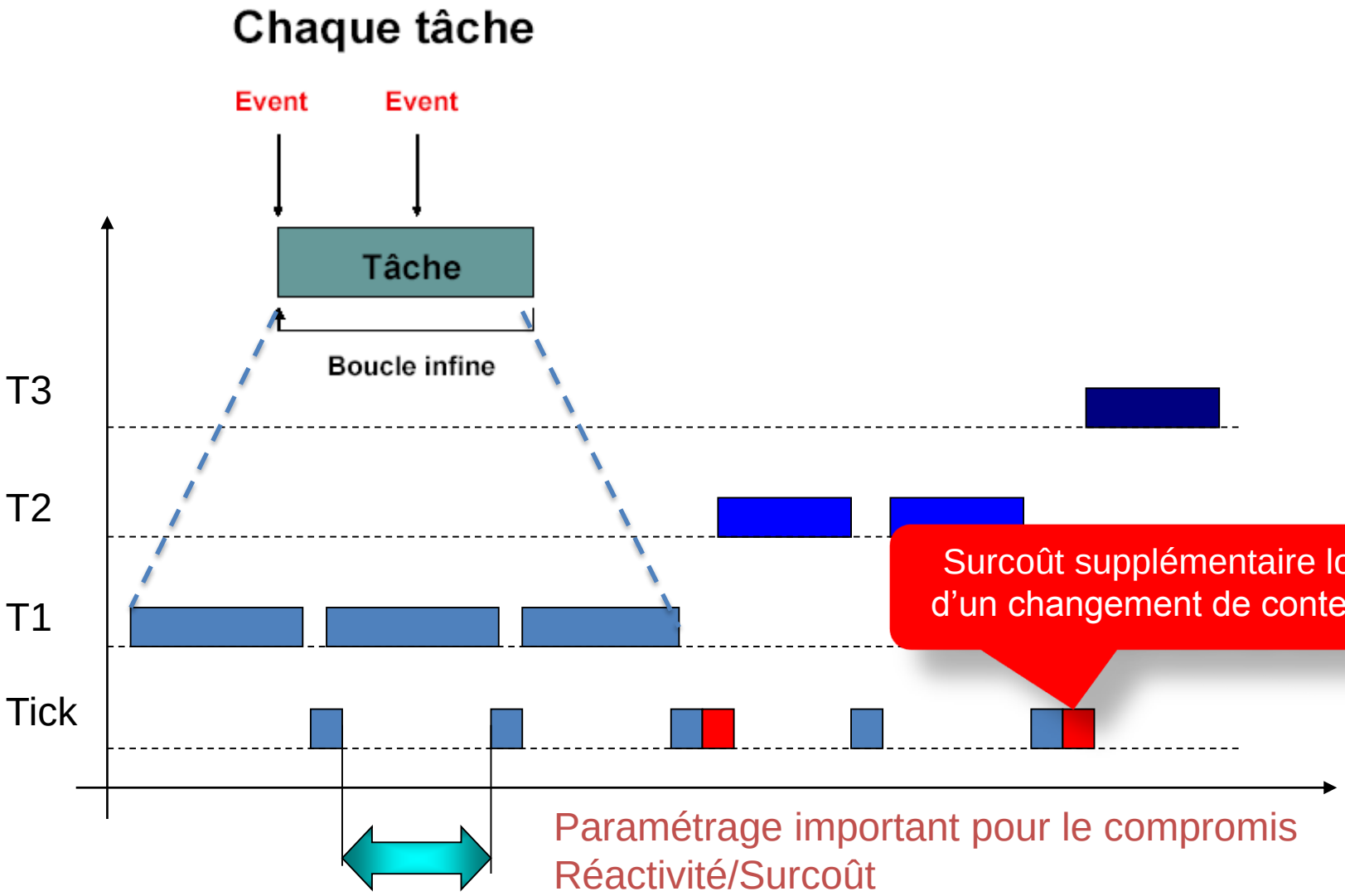
Priorités et ordonnancement



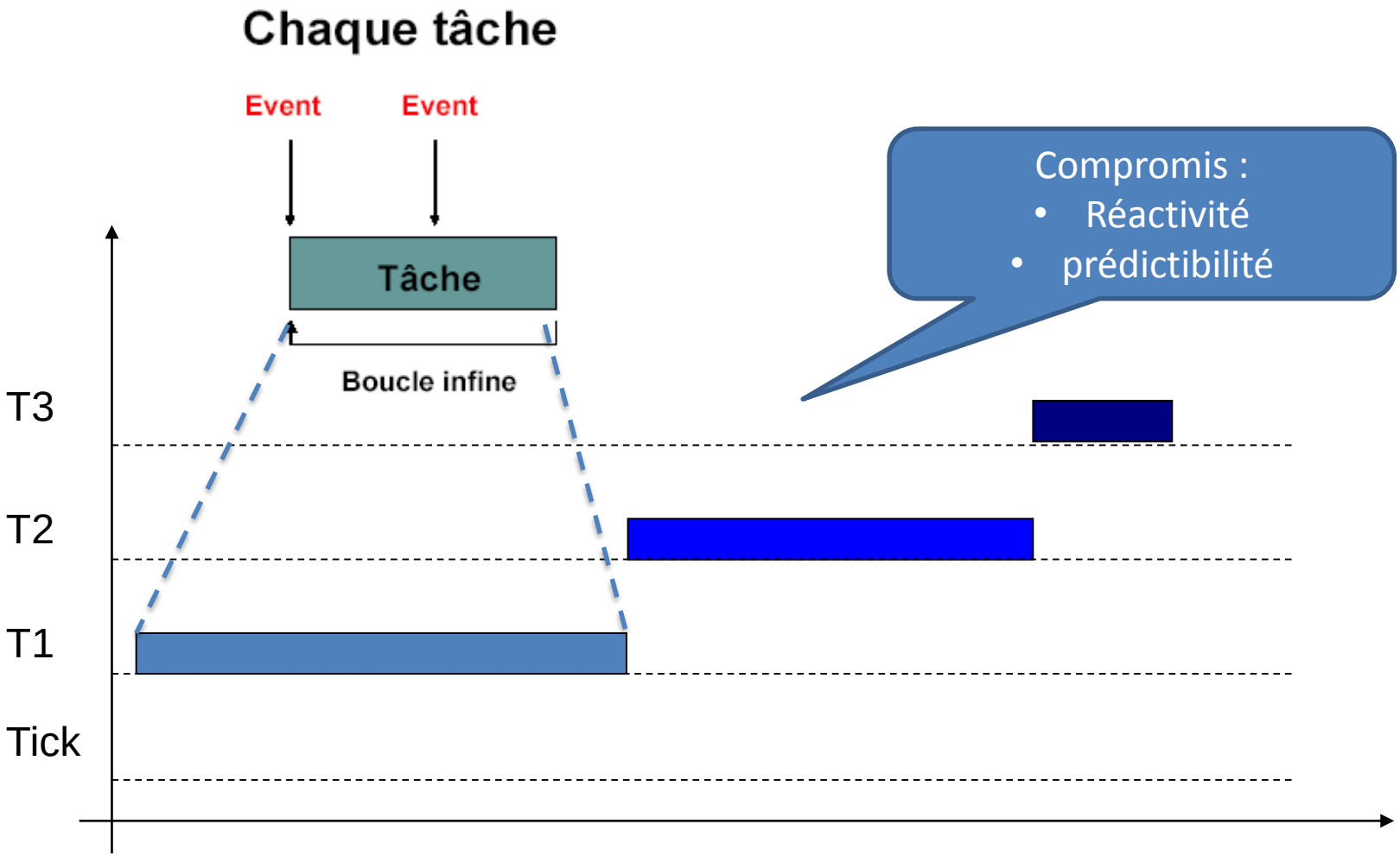
OS Pré-emptif



OS pré-emptif à priorités fixes



OS Non pré-emptif



Fonctions ré-entrantes

Tout appel de fonction par un processus s'exécute dans le contexte du processus.

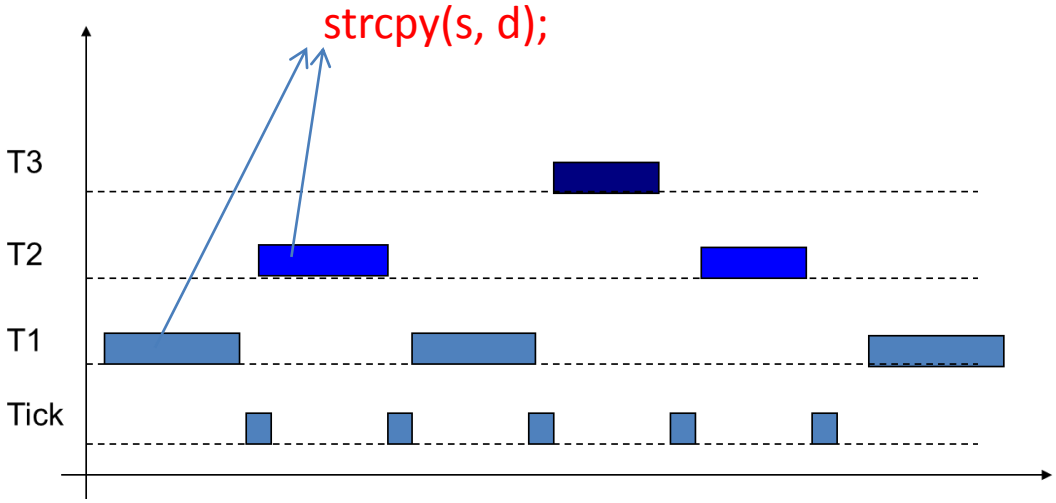
Deux types de fonctions :

- Réentrantes :

```
void strcpy(char *dest, char *src){
    while(*dest++ = *src++);
    *dest = NULL;
}
```

- Non-réentrantes :

```
int temp;
void swap(int *x, int *y){
temp = *x;
*x = *y;
*y = temps;
}
```



Mot-clé volatile

- Fonction inline :
  - Principe identique à une macro du préprocesseur
  - Le compilateur ne génère pas de symbôle associé mais intègre le code de la fonction dans l'appelant
- Variable volatile :
  - Précise au compilateur que la valeur de la variable peut changer à n'importe quel moment sans action explicite du code (registres mappés mémoire, variable modifiée par intérruptions, application multi-thread)
  - Utilisé surtoût en contexte embarqué

Exemple 1 : Registres périphériques

- Exemple d'un registre 8-bit mappé à l'adresse 0x1234. Le code doit tester s'il passe à non-zéro
  - UINT *ptr = (UINT *) 0x1234;
  - // wait until non-zero
  - While (*ptr==0)
  - // do treatment
- Le compilateur génère
  - mov ptr, #0x1234
  - mov a, @ptr
  - loop :
  - bz loop // infinite loop
- En déclarant ptr volatile (UINT volatile *ptr), sa valeur est relue
  - mov ptr, #0x1234
  - loop :
  - mov a, @ptr
  - bz loop

Exemple 2 : routine d'interruption

- Teste de fin de message (ETX) sur un port série
- int etx_rcvd = FALSE;
- void main(){
  - …
  - while (!etx_rcvd){
  - // wait
  - }
  - // unused code
- }
- interrupt void rx_isr(void){
  - …
  - if (ETX == rx_char)
    - etx_rcvd = TRUE;
- }
- Problème : le compilateur ne voit pas le changement possible de etx_rcvd par l'ISR, la boucle d'attente est donc considérée comme toujours vraie.
- Le code après la boucle est simplement supprimé du code compilé !!

Exemple 3 : préemption et variables partagées

- Le compilateur n'a pas de vision des changements de contexte,
- Les variables globales partagées (même par mutex) peuvent donc changer inopinément
- Donc elles doivent être déclarées volatile !

```
int cntr;
void task1(){
        cntr = 0;
        while (cntr==0){
                sleep(1);
        }
…
}
void task2(){
        …
        cntr++;
        sleep(10);
        …
}
```

Inter-Process Communication : IPC

Communication entre tâches
i) Ressource partagée

- Le moyen le plus simple pour faire communiquer 2 tâches est d'utiliser une zone de mémoire partagée.
- Surtout lorsque celles-ci s'exécutent dans un espace d'adressage unique.
- Cela nécessite par contre de s'assurer de l'accès exclusif par chaque tâche à un instant donné. Ce qui se traduit par plusieurs méthodes :
  - a) Arrêter les interruptions, // SECTION CRITIQUE
  - b) Test-And-Set operation
  - c) Stopper l'ordonnanceur,
  - d) Utiliser un sémaphore,

a) Sections critiques

- Pour éviter que d'autres tâches ou ISR modifient une section critique, il est nécessaire d'interrompre les interruptions.
- temps d'arrêt des interruptions = paramètre important d'un RTOS (*interrupt latency*)
- Dépend du processeur
- Contenu donc dans 2 macros de OS_CPU.H :
  - OS_ENTER_CRITICAL()
  - // Section critique
  - OS_EXIT_CRITICAL()

Interruptions

- Les interruptions sont des évènements asynchrones déclenchés par des mécanismes matériels.
- Lorsque le CPU recoit une interruption, il sauvegarde le contexte de la tâche en cours et se branche sur la routine correspondante au numéro d'IRQ dans son vecteur d'interruption.
- A la fin de la routine, le CPU revient à :
  - La tâche la plus prioritaire (mode préemptif)
  - La tâche interrompue (mode non-préemptif)

Interrupt latency

- Une des caractéristiques les plus importantes d'un RTOS = le temps pendant lequel les interruptions sont stoppées (sections critiques).
- Les IRQ sont utilisés en environnement TR embarqué comme moyen de lancement de code utilisateur déclenché par un capteur asynchrone extérieur (freinage ABS).

b) Opération TAS

- Cette opération est utilisée lorsque le système ne dispose pas de noyau TR.
- Principe :
- Test de la valeur d'une variable globale
- Si Val = 0
  - La fonction a accès à la ressource
  - Elle place la variable à 1 // Test and Set
- Certains processeurs implémentent ce service en matériel (instruction TAS du 68000)

c) Stopper l'ordonnanceur

- Solution brutale qui n'interrompt par pour autant les interruptions. Exemple :
- OSSchedLock()
- Access to share data
- OSSchedUnlock()

d) Sémaphores

- Inventés dans les années 1960 par Edgser Dijkstra
- Binary semaphores // Mutex
- Counting semaphores
- Trois types d'opérations :
  - Create (initialize)
  - Wait (Pend) (sem>0)?sem --:wait();
  - Signal (Post) (sem==0)?sem ++ : notify();
- La tâche qui en lancée (notify) est soit
  - La plus prioritaire (uC/OS)
  - Celle qui l'a demandée en premier (FIFO)

Bilan de l’accès en ressource partagée

- Pour l’accès à une zone partagée, le sémaphore est la solution la moins risquée.
- Si les autres solutions sont mal utilisées, les conséquences peuvent être beaucoup plus graves.
- Cependant, pour un simple accès à une variable 32 bits, l’arrêt des interruptions sera moins coûteux que l’utilisation d’un sémaphore sans modifier le ‘*interrupt latency*’ !

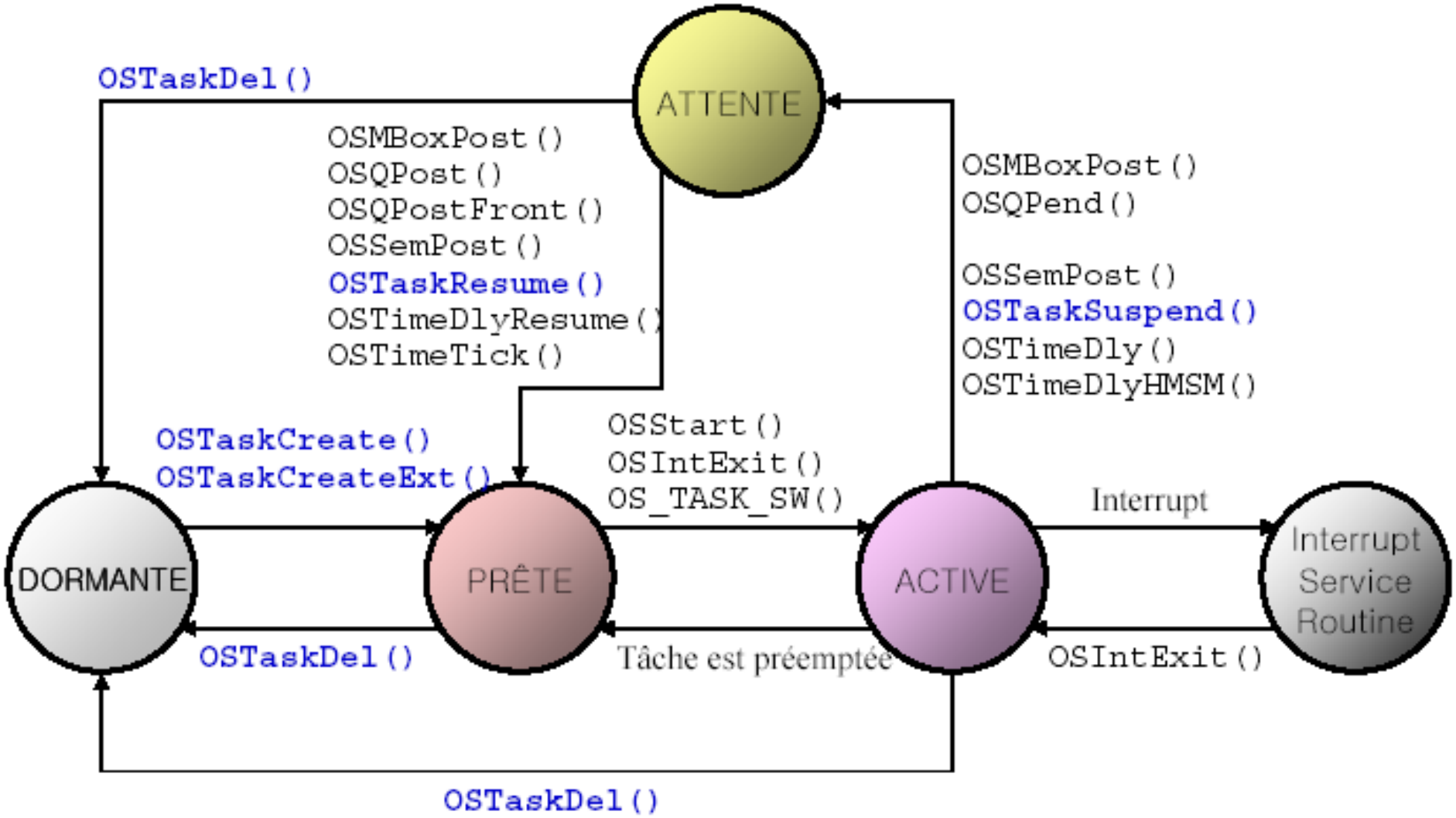
Deadlock

- Le deadlock se produit lorsque 2 tâches attendent l'accès à des ressources prises par l'autre (cf. cours sur l'ordonnancement).
- Pour éviter ces situations les tâches doivent :
  - Acquérir les ressources dans le même ordre,
  - Relâcher les ressources dans l'ordre inverse.
- Les noyaux de RTOS permettent également de spécifier un Timeout sur l'attente des sémaphores.
- Ces appels retournent un code d'erreur différent pour informer la tâche que l'attente ne s'est pas déroulée correctement.

ii) Envoie de messages

- Message Mailbox
  - Envoie d'un pointeur dans une boîte à lettre
  - Une seule lettre à la fois !!
- Message Queue
  - Envoie de plusieurs messages (mailbox queue[] ;)
  - Lecture en mode FIFO
- Une liste de processus en attente est construite par l'ordonnanceur.

Diagramme d'état des processus dans uC/OS-II



Autres services

Les types simples

- Les entiers dépendent de l'architecture cible, ils sont donc redéfinis dans la partie spécifique : dans OS_CPU.h
- Sur PC,
  - typedef unsigned char BOOLEAN;
  - typedef unsigned char INT8U; /* Unsigned 8 bit quantity */
  - typedef signed char INT8S; /* Signed 8 bit quantity*/
  - typedef unsigned int INT16U; /* Unsigned 16 bit quantity */
  - typedef signed int INT16S; /* Signed 16 bit quantity*/
  - typedef unsigned long INT32U; /* Unsigned 32 bit quantity */
  - typedef signed long INT32S; /* Signed 32 bit quantity */
  - typedef float FP32; /* Single precision floating point */
  - typedef double FP64; /* Double precision floating point */

Les délais

- uCOS travaille en temps réel grâce à la notion qu'il se fait du temps.
- Ce temps est donné par une source appelé Clock Tick qui est une interruption (ISR) périodique.
- Dépend de l'application.
- Règle 3 : ISR = overhead par rapport au système
  - trop petit et les tâches prioritaires attendront plus longtemps
  - trop grand et le surcoût deviendra un handicap

OSTimeDly()

- Une tâche peut elle-même se mettre en attente
- L'appel de cette fonction cause un changement de contexte vers la prochaine tâche prioritaire
- Paramètre = nombre de tick entre 0 et 65,535
- La tâche ne s'exécutera à nouveau qu'après le temps écoulé et seulement si elle est la plus prioritaire.

- OSTimeDlyHMSM() fait la même chose en prenant en paramètre des Heures, Minutes, Secondes et Millisecondes.
- Règle 4 : Ne jamais appeler cette fonction après avoir désactivé les ISR!!

OSStart()

- Elle initialise les variables et les structures de données
- Elle crée la tâche Idle() toujours prête à s'exécuter et de priorité la plus faible (OS_LOWEST_PRIO = 63)
- Si les tags OS_TASK_STAT_EN et OS_TASK_CREATE_EXT sont placés à 1, elle crée aussi une tâche de statistique de priorité OS_LOWEST_PRIO -1

OSStart()

- Lance la gestion multi-tâche
- Règle 1 (rappel) : Vous devez donc avoir déjà créer au moins une tâche
- OSStart trouve le TCB de la HPT
- Elle appelle OSStartHighRdy() qui est décrite dans OS_CPU.asm en fonction du processeur cible.

Résumé

Schéma général d'un programme sous uCOS

```
// 1 - Allocation statique des piles d'exécution
OS_STK TaskStartStk[TASK_STK_SIZE];

// 2 - Déclaration des services de communication, synchronisation
OS_EVENT *mbox, mq, mutex;

void main (void){

// 3 - Create at least one task
OSTaskCreate(   TaskStart,
                                (void)*0,
                                &TaskStartStk[TASK_STK_SIZE -1],
                                TASK_START_PRIO);
OSStart();
}

void TaskStart(void *pdata){
        // 4 - Appel/Création des autres tâches de l'application
}
```