

I Introduction sur les Systèmes Embarqués

II Les outils de développement

III Interaction avec l'environnement : les transfert de données

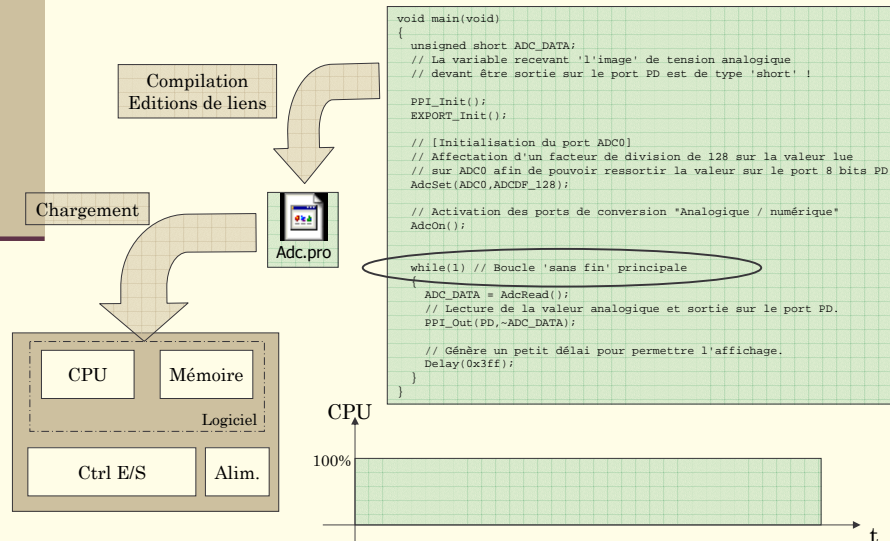
IV Interaction avec l'environnement : les interruptions

V Temps Réel

- 1 Les notions de base en « Temps Réel »
- 2 Les synchronisations : les signaux logiciels et les sémaphores
- 3 Les communications : les échanges de données
- 4 L'interblocage
- 5 L'ordonnancement
- 6 La réalisation d'un RTOS
- 7 Les OS du marché

- Une application est dite « Temps Réel » lorsqu'elle est capable de « réagir » aux évolutions de l'environnement.
- Application est suffisamment performante pour traiter à la volée et sans perte d'information toutes les données provenant d'un ou plusieurs capteurs, sans atteindre la capacité maximale de calcul qu'elle a à sa disposition.
- Dans la plupart des cas, les données sont traitées suffisamment rapidement pour permettre l'obtention des résultats en un temps inférieur à la constante de temps régissant l'évolution du système perçu par les capteurs.
= **Temps Réel « Mou »**
- Une application est dite « Temps Réel » lorsqu'on a la garantie qu'une tâche donnée sera exécutée « dans le pire des cas » en une durée connue et bornée.
= **Temps Réel « Dur »** (déterministe)

Application = 1 code source unique = 1 seule « tâche »



Application = plusieurs « tâches » = plusieurs fonctions (ou « main »)



Une tâche par « Virtual Processor Unit »

```
// Tâche N° 1 (initialisation)
#define HEAP_INIT_COMPLETE 0 VPU1

void main(void)
{
    // On indique que les ports ne sont pas initialisés
    HeapWrChar(HEAP_INIT_COMPLETE, FALSE);

    // Initialisation des ports
    PPI_Init();
    EXPORT_Init();

    // On indique que les ports ont été initialisés
    HeapWrChar(HEAP_INIT_COMPLETE, TRUE);
}
```

```
// Tâche N° 2
#define HEAP_INIT_COMPLETE 0 VPU2

void Delay(int pTIME)
{
    while(pTIME--);
}

void main(void)
{
    Delay(0xff);
    // Attend ' le feu vert' de la tâche N°1
    while(!HeapRdChar(HEAP_INIT_COMPLETE));
    Delay((int)0xff*1/8);
    while(1)
    {
        PPI_BitOut(PD, 0, ~PPI_BitIn(PD, 0));
        Delay(0xff);
    }
}
```

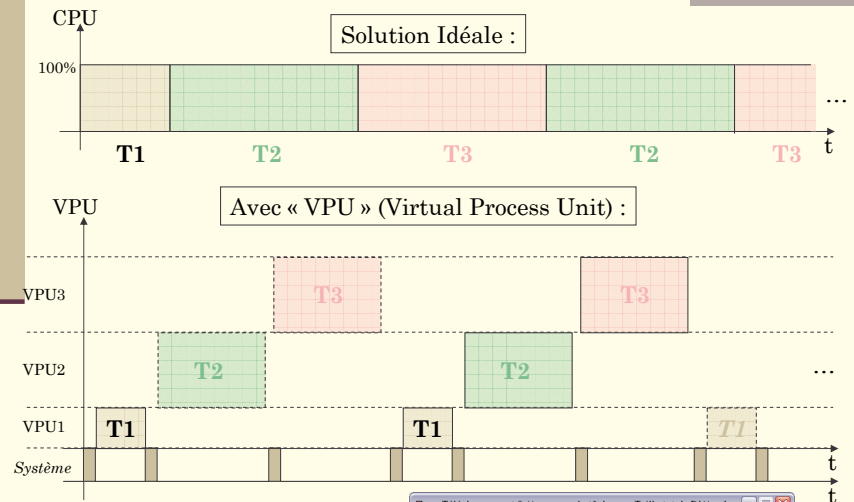
E. Mesnard

```
// Tâche N° 3
#define HEAP_INIT_COMPLETE 0 VPU3

void Delay(int pTIME)
{
    while(pTIME--);
}

void main(void)
{
    Delay(0xff);
    // Attend ' le feu vert' de la tâche N°1
    while(!HeapRdChar(HEAP_INIT_COMPLETE));
    Delay((int)0xff*2/8);
    while(1)
    {
        PPI_BitOut(PD, 1, ~PPI_BitIn(PD, 1));
        Delay(0xff);
    }
}
```

5



T1 : VPU1 = 20% mémoire
T2 : VPU2 = 40% mémoire
T3 : VPU3 = 40% mémoire

E. Mesnard

Téléchargement & Manager de tâches : Taille totale RAM = 1...

	Nouveau	Ouvrir	Sauver	Sauver Sous	Ajouter Tâche	Effacer Tâ
0	T1.pro	0	19656	19657	39321	20 %
1	T2.pro	0	39317	39318	78643	40 %
2	T3.pro	0	39317	39318	78643	40 %

NewFile

6

- Exécutif (« Operating System » – Système d'exploitation) :
Nécessaire pour donner l'illusion qu'il y a autant de processeurs que de tâches qui demandent à être exécutées
- L'exécutif comprend :
 - Un ordonnanceur (« scheduler »), chargé d'attribuer le CPU aux tâches
 - Des primitives (fonctions ininterrompibles et indivisibles) : Init, Start, Créer, Fin, ISem, P, V, Atit, ...
- Rôles de l'exécutif (via l'appel ou non de « primitives ») :
 - Gérer les tâches
 - Créer, détruire, ... les tâches,
 - Faire cohabiter différents types de tâches : périodiques ou non, avec différents niveaux de priorités...
 - Ordonnancer (« scheduler ») les tâches pour respecter des échéances,
 - Réaliser l'exécution « parallèle » (pseudo parallèle) des tâches (gérer les changements de contexte,...),
 - Changer (le cas échéant) dynamiquement des priorités.
 - Gérer les interruptions
 - Horloge,
 - IT externes,
 - IT utilisateurs.
 - Synchroniser les tâches entre elles,
 - Messages de synchronisations,
 - Sémaphore (compteurs de consommation).
 - Gérer les communications entre les tâches
 - Echange de données par messages directement en mémoire,
 - Echange de données dans des boîtes aux lettres.
 - Résoudre les conflits d'accès (ressources partagées entre les tâches)
- Et tout ça, en fonctionnant en « Temps Réel »

E. Mesnard

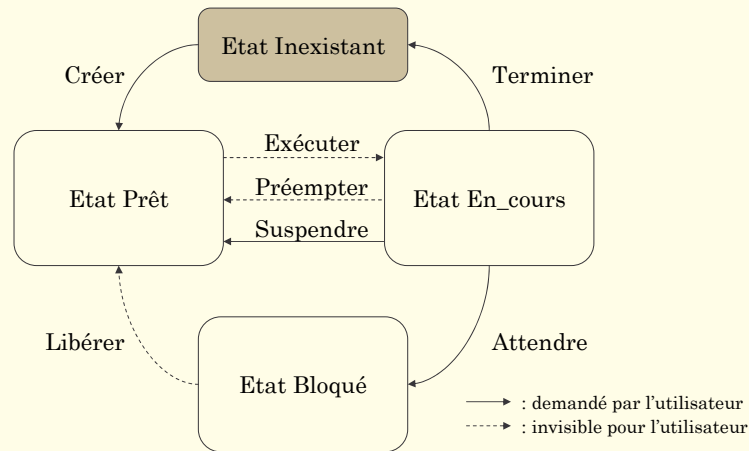
7

- Une application TR = ensemble de tâches : multiprogrammation avec dépendance du temps.
- Une tâche = regroupement des procédures qui doivent être exécutées à la suite de l'apparition du même événement (IT matérielle ou logicielle).
- Une tâche est définie par un descripteur :
 - Pointeur sur la fonction d'entrée
 - Priorité associée à la tâche
 - Contexte de la tâche (registres en cours d'utilisation)
 - Valeur de la période si elle est périodique
- A certains instants, il peut y avoir plusieurs tâches qui désirent être exécutées... il faut en élire une !
- Le choix de l'élue :
 - À la charge d'un « exécutif »
 - Plusieurs politiques :
 - Les tâches actives ont la même priorité
 - Elles ont toutes des priorités différentes
 - Elles sont de priorités différentes et/ou égales
 - Les priorités sont dynamiques

E. Mesnard

8

- Inexistant (Hors_Service) : tâche en mémoire mais non connue de l'exécutif
- Prêt : tâches prêtes à être exécutées
- En_cours : tâche qui est choisie (par l'exécutif) et qui s'exécute sur le processeur
- Bloqué : tâches en attente d'événements (IT périodique, IT externe, ...)



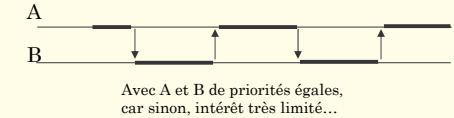
- Suspendre / Préempter :
 - arrêt temporaire de la tâche en exécution,
 - la tâche est remise ainsi dans la liste des tâches prêtes (à la fin).
 - ces primitives provoquent l'action Exécuter, pour qu'une autre tâche prête s'active
- Exemple : faux temps partagé, quasi parallélisme contrôlé par les tâches :

```

Tache_A()
{ while(1) {
  Suspendre();
  ...
} }

Tache_B()
{ while(1) {
  ...
  Suspendre();
} }

```



- Exemple : vrai temps partagé, quasi parallélisme contrôlé le gestionnaire d'interruption, activé par une horloge (ISR de l'IT d'un timer) :

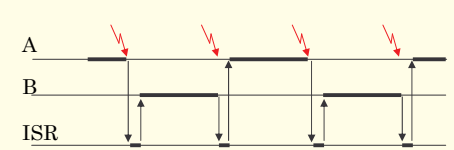
```

ISR()
{ switch (IT) {
  case Tick : Preempter();
  ...
} }

Tache_A()
{ while(1) {
  ...
} }

Tache_B()
{ while(1) {
  ...
} }

```



- Que doit faire l'ordonnanceur quand aucune tâche n'est prête ?

Exécution d'une Tâche de fond :

```

Idle_Task()
{
  while(1);
}

```

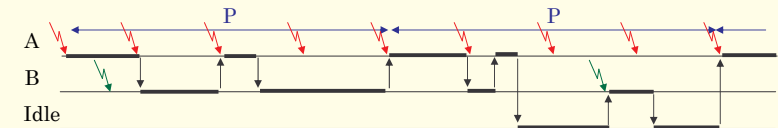
- Cette tâche ne se bloque JAMAIS !
- Donc, elle est toujours prête.
- Obligatoirement la priorité la plus faible.
- Variante(s) sur son code :
 - Elle demande en permanence à l'ordonnanceur de tenter d'exécuter une tâche différente d'elle-même...
 - et cela sans attendre l'interruption périodique de l'horloge du système :

```

Idle_Task()
{
  while(1){
    Suspendre();
  }
}

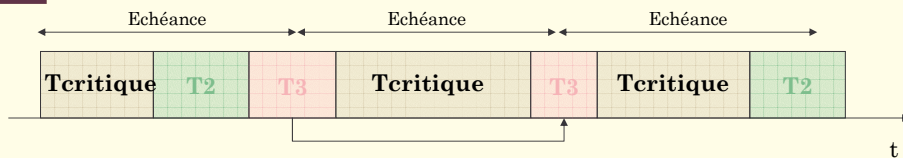
```

- Tâche « Périodique » :
 - la tâche doit s'exécuter de manière périodique,...
 - même si elle est **préemptée** dans l'intervalle de la période.
- Tâche « Sporadique » :
 - Sporadique = Non périodique = apériodique
 - la tâche s'exécute sur un événement particulier, pas nécessairement lié à l'horloge.
- Exemple :
 - Tâche A : périodique de période P=4 (unité : tics d'horloge)
 - Tâche B : sporadique, de même priorité que A



- Nécessité d'une structure de données PERIODES donnant les valeurs des périodes des tâches périodiques (sous la forme de compteurs).
- A chaque IT d'horloge, ces périodes doivent être décrémentées.
- Les tâches dont le compteur devient nul repassent dans l'état Prêt.

- **Tâche critique :**
 - Tâche pour laquelle la boucle de traitement doit se dérouler impérativement dans un intervalle de temps donné sous peine de défaillance du système complet.
 - Échéance : date limite
- **Temps Réel Dur :**
 - S'assurer que les autres tâches n'auront pas la possibilité de venir perturber la tâche critique au point de l'empêcher de faire son travail dans les temps.



Rôle de l'ordonnanceur : assurer le respect des échéances

- **Objectifs des signaux logiciels :**
 - Eviter les attentes actives (boucles de scrutation).
 - Synchroniser les tâches asynchrones par *émission* et *attente* d'un signal.
- **Implémentation :**
 - Un signal, noté S \Leftrightarrow booléen indiquant l'occurrence de l'événement
 - Manipulation du signal à l'aide de primitives :
nom de la fonction « système » appelée
- **Primitive d'émission : `signal(S)` ou `!(S)`**
 - Emission **non bloquante** du signal logiciel S
 - Rend activable **toutes** les tâches bloquées en attente de S.
 - Si aucune tâche en attente, alors le signal est mémorisé (S=1) dans le cas le plus courant, ou bien le signal est perdu (implémentation « fugace »)
 - En résumé, plusieurs émissions successives non attendues
 - en « fugace » = aucune émission
 - en « mémorisé » = une seule émission
- **Primitive d'attente : `wait(S)` ou `?(S)`**
 - Attente **potentiellement bloquante** du signal logiciel S
 - Si S=0, provoque le blocage de la tâche appelante.
 - Si S=1, alors S est remis à 0 et la tâche appelante reste active.

- 4 tâches (T1, T2, T3 et T4) asynchrones (en boucle) qu'il faut synchroniser : d'abord T1 (au moins une fois), puis T2 et T3 en parallèle, et enfin T4, ... et ainsi de suite.
- **Solution :**
 - 3 signaux (à mémorisation simple) : S1, S2 et S3 pour gérer les transitions

```

main()
{
  creer(Tache_T1);
  creer(Tache_T2);
  creer(Tache_T3);
  creer(Tache_T4);
  executer();
}

Tache_T1()
{
  while(1) {
    .
    .
    .
    signal(S1);
  }
}

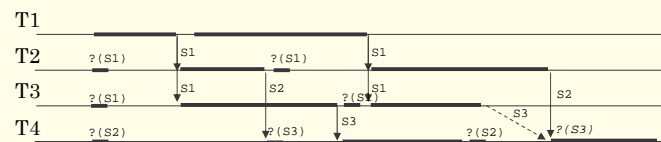
Tache_T2()
{
  while(1) {
    wait(S1);
    .
    .
    .
    signal(S2);
  }
}

Tache_T3()
{
  while(1) {
    wait(S1);
    .
    .
    .
    signal(S3);
  }
}

Tache_T4()
{
  while(1) {
    wait(S2);
    wait(S3);
    .
    .
    .
  }
}

```

Trace possible d'exécution :



Attention : implémentation « hasardeuse »...
« puis T2 et T3 en parallèle » : OK, mais combien d'instances ?

- 2 tâches (T1, T2) asynchrones qu'il faut « bi-synchroniser » :
Bi-synchronisation : principe de Rendez-vous
Les tâches se donnent rendez-vous à un point particulier dans leur code, puis elles continuent en parallèle leur traitement.
- **Solution :**
 - 2 signaux en attente croisée

```

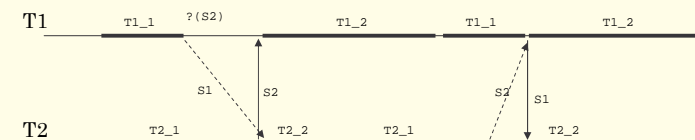
main()
{
  creer(Tache_T1);
  creer(Tache_T2);
  executer();
}

Tache_T1()
{
  while(1) {
    . // T1_1
    .
    .
    signal(S1);
    wait(S2);
    . // T1_2
    .
    .
  }
}

Tache_T2()
{
  while(1) {
    . // T2_1
    .
    .
    signal(S2);
    wait(S1);
    . // T2_2
    .
    .
  }
}

```

Trace possible d'exécution :



- Principe des sémaphores de Dijkstra (1965)
- Objectifs :
 - Synchroniser les tâches, **mais** contrairement aux signaux logiciels, ne libérer qu'un nombre donné de tâches.
 - Protéger les ressources partagées utilisées par au moins deux tâches asynchrones concurrentes
- Implémentation :
 - Le sémaphore (événement logiciel) est un mécanisme basé sur une **variable entière partagée**.
 - Possibilité de créer autant de sémaphores que nécessaire.
- Comptabilisation :
 - Contrairement aux signaux logiciels, le **nombre d'appels des primitives** joue un rôle.

- Création et initialisation de la variable associée au sémaphore :
 - `Init_Sem(Semaphore, Valeur)` ou `Vn(Semaphore, Valeur)`

Cette fonction réalise : `Semaphore = Valeur;`
 - La valeur d'un sémaphore est modifiée ensuite par deux primitives de l'exécutif :
 - **V(Semaphore)** :
émission **non bloquante** de l'événement logiciel

Cette fonction réalise : `Semaphore = Semaphore + 1;`
Puis, l'exécutif rend la main à la fonction qui a appelé cette primitive.

 - **P(Semaphore)** :
attente **potentiellement bloquante** de l'événement logiciel
- Cette fonction réalise **d'abord** : `Semaphore = Semaphore - 1;`
Puis, l'exécutif **bloque** la tâche si cette valeur du sémaphore est strictement négative. Il redonne alors la main à une autre tâche prête (si aucune tâche utilisateur, alors, exécution de la tâche de fond).

- 2 tâches (T1, T2) asynchrones qu'il faut synchroniser ainsi :
T1 et T2 sont constituées de deux parties T1_1 et T1_2, et T2_1 et T2_2.
une première partie (T2_1) de la tâche T2 doit systématiquement s'être exécutée avant la seconde partie (T1_2) de la tâche T1 (afin que cette dernière puisse se réaliser correctement)

- Solution :

- Un sémaphore S, initialisé à 0

```

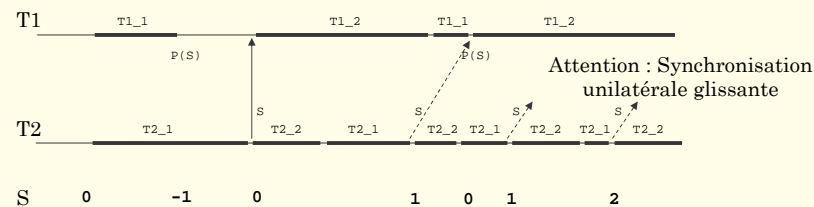
main()
{
  creer(Tache_T1);
  creer(Tache_T2);
  Vn(S,0);
  executer();
}

Tache_T1()
{
  while(1) {
    // T1_1
    P(S);
    // T1_2
  }
}

Tache_T2()
{
  while(1) {
    // T2_1
    V(S);
    // T2_2
  }
}

```

Trace d'exécution possible :



- Identique exemple 1
- 4 tâches (T1, T2, T3 et T4) asynchrones qu'il faut synchroniser :
d'abord T1, puis T2 et T3 en parallèle, et enfin T4
- Solution :
 - 3 sémaphores : S1, S2 et S3 pour gérer les transitions

```

main()
{
  creer(Tache_T1);
  creer(Tache_T2);
  creer(Tache_T3);
  creer(Tache_T4);
  Vn(S1,0);
  Vn(S2,0);
  Vn(S3,0);
  executer();
}

Tache_T1()
{
  while(1) {
    // T1_1
    P(S1);
    // T1_2
  }
}

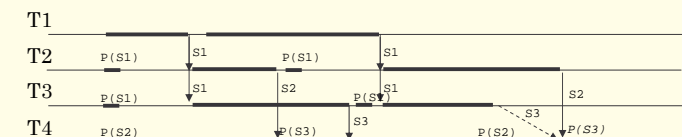
Tache_T2()
{
  while(1) {
    P(S1);
    // T2_1
    V(S2);
    // T2_2
  }
}

Tache_T3()
{
  while(1) {
    P(S1);
    // T3_1
    V(S3);
    // T3_2
  }
}

Tache_T4()
{
  while(1) {
    P(S2);
    P(S3);
    // T4_1
  }
}

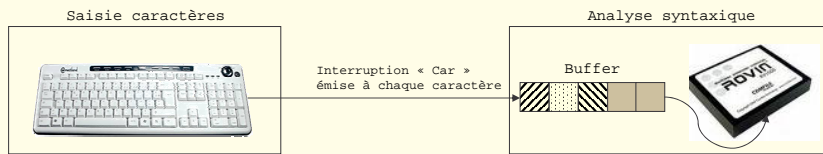
```

Trace d'exécution possible :



■ Objectif de l'application :

- Analyse syntaxique sur des phrases saisies en temps réel sur un clavier



■ Applications constituées de 2 tâches :

- Tâche R : Réception de phrases, caractère par caractère, pour stockage dans un buffer de réception.

La frappe d'un caractère sur le clavier provoque une interruption « Car » sur le système.

La fonction **char Lire_Car()** permet alors de lire le bus de données, pour retourner ainsi le caractère saisi.

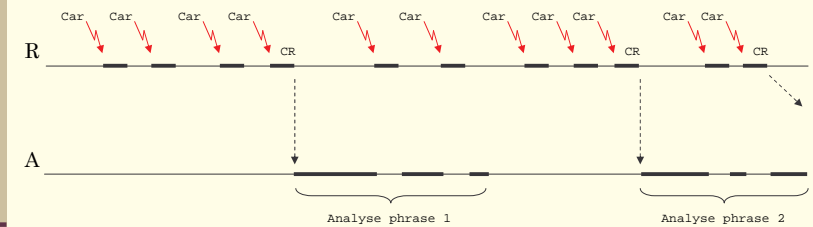
- Tâche A : Analyse syntaxique d'une phrase prélevée dans le buffer.

L'appel de la fonction **Analyse(char *Phrase)** permet de réaliser cette analyse syntaxique.

■ Le buffer est **Buffer[5][80]** :

- 5 blocs pouvant chacun contenir une phrase.
- Phrase : au maximum 80 caractères, et moins, si CR (« Carriage Return ») reçu avant.

Fonctionnement souhaité



- Le CR provoque systématiquement l'analyse d'une des phrases du buffer.
- Au pire, tous les 80 caractères, la phrase sera analysée.
- Une nouvelle phrase peut être intégralement saisie alors que l'analyse de la précédente n'est pas terminée.

■ Variables et sémaphores d'une solution à cet exemple :

■ Variables :

- Index : entier, initialisé à 0, représentant la position du caractère dans la phrase. Cet entier sera incrémenté à chaque nouvel réception, jusqu'à la valeur maximale 79.
- Buf_E : entier, initialisé à 0, indiquant le numéro de buffer utilisé en Ecriture.
- Buf_L : entier, initialisé à 0, indiquant le numéro de buffer utilisé en Lecture.
- Ces deux variables Buf_E et Buf_L seront incrémentées en modulo 5.

■ Sémaphores :

- **Caractere** : sémaphore, initialisé à 0, indiquant qu'une IT de réception d'un caractère est survenue
 ⇒ V(Caractere) dans l'ISR de traitement de l'IT « Car »
 ⇒ P(Caractere) juste avant de lire un caractère sur le port de données
- **Deb_Analyse** : sémaphore, initialisé à 0, permettant de lancer l'analyse.
 ⇒ P(Deb_Analyse) en début de la tâche A
- **Blocs_Libres** : sémaphore, initialisé à 5, représentant le nombre de blocs libres dans le buffer.
 ⇒ P(Blocs_Libres) en début de la tâche R

Source de l'exemple :

```

// V-2 Exemple 5 - les sémaphores pour synchr
// E. Mesnard (c) ISIMA 2009/2010

#include "RTOS.h";
#define CR 13; // Carriage Return

char Buffer[5][80]; // Buffer pour stocker les phrases
int Buf_L, Buf_E; // Indices de lecture / écriture dans le buffer
int index; // Position du caractere dans le buffer
char C_lu; // Caractere lu
int Caractere, // semaphore de presence de caracteres
Debut_Analyse, // semaphore de synchro pour debuter l'analyse
Blocs_Libres; // semaphore de comptage de blocs restants dans le buffer

void main()
{
    creer(Tache_A);
    creer(Tache_B);
    Vn(Caractere,0); // Initialisation des sémaphores
    Vn(Debut_Analyse,0);
    Vn(Blocs_Libres,5);
    Buf_L=0; // Initialisation des pointeurs
    Buf_E=0; // sur les buffers
    index=0;
    executer();
}

void ISR(void) // Interrupt Service Routine
{
    unsigned char EVT;

    EVT=GetMsg(); // Lecture d'un evenement
    switch(EVT)
    {
        case MSG_IT_Car : // Interruption Car survenue
            V(Caractere); // Signale aux taches qu'un caractere est survenu
            break;
        default : // En cas d'IT étranges...
            DebugPrint("\n !!! IT non prévue : ");
            break;
    }
}

```

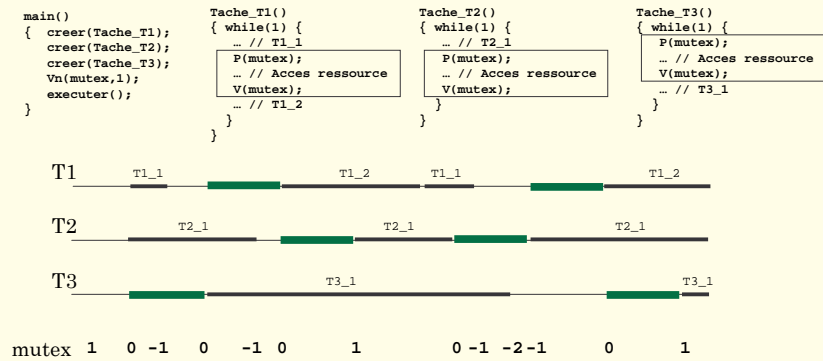
V-2 Exemple 5 – les sémaphores pour synchroniser et compter

```
Tache_R() // Tache de reception des caracteres
{ while(1) {
    P(Blocs_Libres);          // Ne traite rien tant qu'il n'y a plus de place
    index=0;
    C_Lu='';
    while ( (C_Lu!=CR) && (index!=80) ) { // Tant que pas CR ni 80 carac.
        P(Caractere);          // attente d'un caractere, via l'IT
        C_Lu = Lire_Car();
        Buffer[Buf_E][index] = C_Lu;    // Memorisation dans le buffer
        index++;
    }
    // Changement de phrase (bloc de buffer)
    Buf_E=(Buf_E==4) ? 0 : Buf_E++;    // incrementation modulo 5
    V(Debut_Analyse);                // lancer une analyse
} }
```

```
Tache_A() // Tache d'analyse syntaxique d'une phrase
{ while(1) {
    P(Debut_Analyse);          // attente synchro debut d'analyse
    Analyse(Buffer[Buf_L]);    // analyse du buffer correspondant
    Buf_L=(Buf_L==4) ? 0 : Buf_L++; // incrementation modulo 5
    V(Blocs_Libres);          // signaler que le bloc est traite, donc libre
} }
```

V-2 Exemple 6 – les sémaphores pour protéger

- 3 tâches (T1, T2, T3) asynchrones qui utilisent une ressource partagée :
A tout moment, la ressource ne peut être utilisée que par 1 seule tâche
T1 : T1_1, puis utilisation de la ressource, T1_2
T2 : T2_1, puis utilisation de la ressource
T3 : utilisation de la ressource, puis T3_1
- Solution :
 - 1 sémaphore mutex, initialisé à 1, pour provoquer une exclusion mutuelle



V-2 Exemple 7 – les sémaphores pour synchroniser en Rendez-Vous

- Même type de problème que l'exemple 2
- Rendez-vous généralisé à (n+1) tâches (multi-synchronisation) :
 - 1 tâche Tache_RdV doit attendre que n autres tâches Ti soient parvenues à un endroit précis pour poursuivre son exécution.
 - De même, ces tâches Ti ne peuvent poursuivre que si la tâche Tache_RdV est effectivement au point de rendez-vous.
- Une solution possible est :
 - Une variable partagée entière nb, initialisée à 0, comptabilisant le nombre de tâches au point de rendez-vous.
 - un sémaphore mutex, initialisé à 1, pour protéger la variable partagée nb
 - un sémaphore RdV_attente, initialisé à 0, pour le blocage de la tâche RdV au point de rendez-vous
 - un sémaphore Tache_attente initialisé à 0, pour le blocage des n tâches

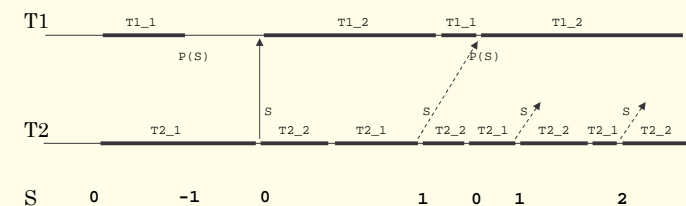
```
#define n ...
int nb;
main()
{
    creer(Tache_T1);
    ...
    creer(Tache_Ti);
    ...
    creer(Tache_Tn);
    Vn(mutex,1);
    Vn(RdV_attente,0);
    Vn(Tache_attente,0);
    nb=0;
    executer();
}
```

```
Tache_RdV()
{
    while(1) {
        ... // T1_1
        P(RdV_attente); // Point de RdV
        Vn(Tache_attente,n); // Liberation
        ... // T1_2
    }
}
```

```
Tache_Ti()
{
    while(1) {
        ... // Ti_1
        P(mutex);
        nb = nb + 1;
        if (nb == n) {
            V(RdV_attente);
            nb = 0;
        }
        V(mutex);
        P(Tache_attente);
        ... // Ti_2
    }
}
```

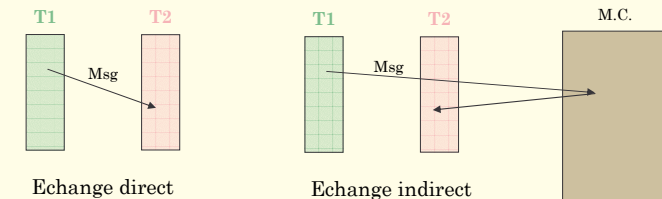
V-2 En résumé : les sémaphores pour la synchronisation

- Synchronisation \Rightarrow Initialisation à la valeur 0
- v(S)** : voisin de **signal(S)**
 - S++
 - rend active **UNE** tâche bloquée si $S \leq 0$ (contrairement aux signaux logiciels pour lesquels signal(S) libère **toutes** les tâches bloquées en attente du signal)
 - Si aucune tâche n'est bloquée, S est seulement incrémenté.
- P(S)** : voisin de **wait(S)**
 - S--
 - blocage de la tâche appelante si $S < 0$.
 - sinon, la tâche reste active.
- Valeur mémorisée :
 - Si $S < 0$: |S| est le nombre de tâches bloquées en attente de S.
 - Sinon : S indique l'excédent d'appels à la primitive V(S) par rapport à P(S). Dans ce cas, S représente le nombre d'événements émis et non consommés.



- Protection des ressources :
 - variables partagées,
 - ressources physiques, ...
- Cette protection réalise alors l'**exclusion mutuelle**.
- Protection \Rightarrow Initialisation du sémaphore de protection, souvent appelé « mutex », à la valeur 1
- Dans chaque tâche manipulant la ressource partagée :
 - Définir la Section critique : zone de code utilisant la ressource partagée
 - Encadrer cette section critique avec le sémaphore :
 - $P(\text{mutex})$: primitive à placer au **début** de la section critique.
 - $V(\text{mutex})$: primitive à placer à la **fin** de la section critique.
- Taux d'utilisation de la ressource partagée :
 - Si $\text{mutex}=1$: aucune tâche n'utilise la ressource
 - Si $\text{mutex}=0$: la ressource est en cours d'utilisation
 - Si $\text{mutex}<0$: $|\text{mutex}|$ est le nombre de tâches en attente de la libération de la ressource partagée.

- Principe des échanges
 - Tâches concurrentes et coopérantes placées sur le même processeur.
 - Le processeur accède, quand il le souhaite, à sa mémoire vive.
 - La mémoire est dite « mémoire partagée », puisque chaque tâche peut alors y accéder.
 \Rightarrow données échangées à travers cette mémoire partagée, de manière **indirecte**.
 - Utilisation d'une zone mémoire dédiée aux échanges = boîte aux lettres



- Avantages à ce mécanisme d'échanges indirects :
 - souplesse dans l'écriture du code
 - même message éventuellement lu par différentes tâches
- Deux primitives (avec plusieurs implémentations possibles...) :
 - **Send(Msg_Box,Msg)** : émission (généralement **non bloquante**)
 - **Receive(Msg_Box)** : réception (**potentiellement bloquante**)

Implémentation « simpliste » :

- Accès directs sur la mémoire
 - Send = write
 - Receive = read
- Exemple :
 - T1 envoie un message à T2

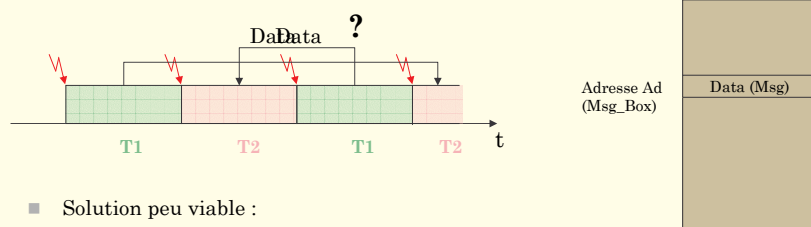
```

main()
{
  creer(Tache_T1);
  creer(Tache_T2);
  executer();
}

Tache_T1()
{
  while(1) {
    ...
    send(Msg_Box,Msg); // write(Ad,Data)
    ...
  }
}

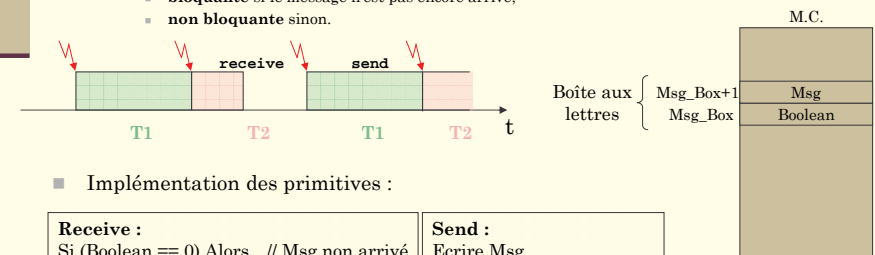
Tache_T2()
{
  while(1) {
    ...
    Data_Rcv=receive(Msg_Box); // read(Ad)
    ...
  }
}

```



- Solution peu viable :
 - Indéterminisme : ordre temporel de lecture/écriture joue un rôle
 - Incohérence potentielle des données lues, donc, des traitements futurs

- Objectif : ajouter un mécanisme de synchronisation aux échanges
 - Rendre les échanges de données **déterministes**
 - Implémentation de la synchronisation avec l'équivalent d'un signal logiciel : booléen de présence du message dans la boîte
- Primitives :
 - **Send_Signal(Msg_Box,Msg)** : émission **non bloquante** du message Msg sur la boîte aux lettres Msg_Box
 - **Msg_Rcv = Receive_Signal(Msg_Box)** : réception sur la boîte aux lettres...
 - **bloquante** si le message n'est pas encore arrivé,
 - **non bloquante** sinon.



- Implémentation des primitives :

Receive :
 Si (Boolean == 0) Alors // Msg non arrivé
 Bloquer tâche
 Sinon // Msg arrivé
 Boolean = 0 // consommation
 Retourner donnée Msg

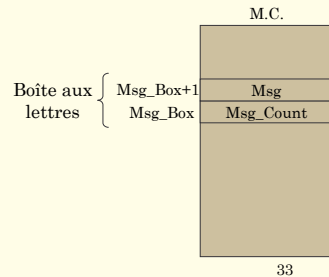
Send :
 Ecrire Msg
 Boolean = 1
 Libérer tâche(s) en attente

V-3 Communication synchronisée en « sémaphore »

- Objectif : comptabiliser les nombres d'émission et de réception
 - contrôler le nombre de lecture qu'il est possible de faire
 - ne pas libérer toutes les tâches en attente d'un coup (comme en « signal »)
 - surveiller d'éventuelles dérives consommation/émission
- Primitives :
 - Send_Sem(Msg_Box,Msg)** : émission **non bloquante** du message Msg sur la boîte aux lettres Msg_Box
 - Msg_Rcv = Receive_Sem(Msg_Box)** : réception sur la boîte aux lettres...
 - bloquante** si le message n'est pas encore arrivé,
 - non bloquante** sinon.
- Implémentation des primitives :
 - Ne pas utiliser un booléen de présence, mais un entier de comptage : Msg_Count
 - Msg_Count est le **sémaphore** du message : nombre de fois que le message peut être lu

Send :
Ecrire Msg
Msg_Count++
Libérer une éventuelle tâche en attente

Receive :
Si (Msg_Count <= 0) Alors // Msg non arrivé
 Bloquer tâche // ou trop consommé
Sinon // Msg arrivé
 Msg_Count-- // consommation
 Retourner donnée Msg

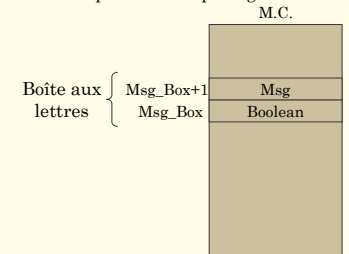


V-3 Communication synchronisée en « Rendez-vous »

- Objectif : bi-synchroniser l'émetteur avec son récepteur
 - éviter toutes les éventuelles dérives consommation/émission
 - par contre, l'émission devient bloquante
 - Variante : multi-synchronisation de l'émetteur avec plusieurs récepteurs (Cf. sémaphores en multi-synchronisation)
- Primitives :
 - Send_RdV2(Msg_Box,Msg)** : émission **bloquante** du message Msg sur la boîte aux lettres Msg_Box
 - Msg_Rcv = Receive_RdV2(Msg_Box)** : réception **bloquante** sur la boîte aux lettres
- Implémentation des primitives en bi-synchronisation :
 - Avec signaux logiciels : S1/S2 pour mécanisme de rendez-vous par signaux logiciels
 - Ou avec sémaphores : S1/S2 pour mécanisme de rendez-vous par sémaphores
 - Ou encore avec un Boolean : mécanisme de rendez-vous par variable partagée

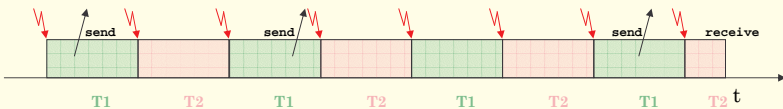
Send :
Ecrire Msg
Boolean = 1 // ou signal(S1), ou V(S1)
Attendre Boolean==0 // ou wait(S2), ou P(S2)

Receive :
Attendre Boolean==1 // ou wait(S1), ou P(S1)
Lire donnée Msg
Boolean=0 // ou signal(S2), ou V(S2)
Retourner donnée Msg



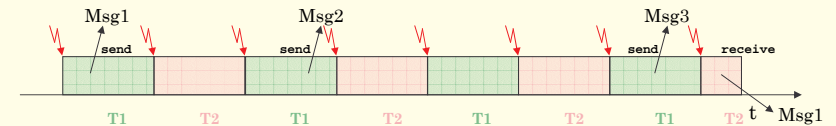
V-3 Analyse : communication par émissions en rafale

- Emissions en rafale :
 - Plusieurs émissions successives, sans réception intermédiaire
- Communication « rendez-vous » :
 - impossible : autant de réception que d'émission
- Communication « signal » :
 - un message non consommé est systématiquement écrasé par le nouveau, donc, totalement ignoré et perdu...
 - Concept : **rafraîchissement d'une donnée**
 - Exemple : capteur mettant à jour la valeur (acquise) dans un registre consulté, de temps en temps, par les tâches qui en expriment le besoin.
- Communication « sémaphore » :
 - un message non consommé est systématiquement écrasé par le nouveau
 - par contre, le compteur de messages émis est incrémenté en conséquence
 - Concept :
 - à chaque **émission**, il doit y avoir un **traitement associé lors de la réception** (de la donnée, plus ou moins rafraîchie, au moment de sa lecture), **sans** la contrainte forte d'une synchronisation totale en **rendez vous**.
 - Exemple : traitement conjoint de tâches coopérantes



V-3 Variantes : communication à mémorisation multiple

- Plusieurs émissions successives = plusieurs messages **différents** à conserver



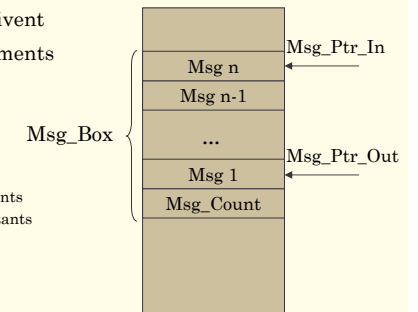
- Plusieurs émetteurs différents sur la même boîte = aussi plusieurs messages **différents** à conserver

- Mémorisation multiple à l'aide d'une file de type « FIFO », située en mémoire partagée.

- Emission / réception (consommation) doivent être équilibrées, pour éviter les débordements de la file.

- Implémentation « aisée »:

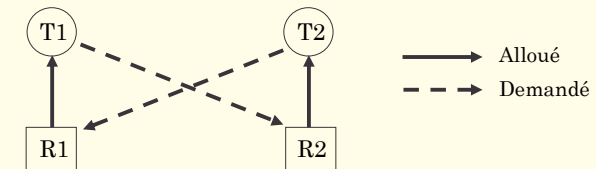
- Communication « sémaphore » et FIFO
- Pointeurs sur la file
 - Msg_Ptr_In : pointeur des messages entrants
 - Msg_Ptr_Out : pointeur des messages sortants



- Problèmes = blocages
- Blocage :
 - Issu des attentes de ressources (wait, P, ...)
 - Attente trop longue, voire même infinie !
- Une sécurité : les « Watchdog » (chien de garde)
 - Watchdog : durée maximale d'attente, après quoi, réveil de la (des) tâche(s)...
- Watchdog = Lourd pour le RTOS :
 - Un chien de garde par objet manipulé : P(Sem, WatchDog), wait(Evt, Watchdog), ...
 - Décrément à chaque tic d'horloge
 - Réveil des tâches en attente quand expiration du délai de garde
 - Informer la (les) tâche réveillée que réveil via le chien de garde
 - Etat PRET_Normal
 - état PRET_Watchdog
 - Gestion différente dans la tâche selon le mode de réveil
- Une autre sécurité : éviter/corriger les attentes infinies dues aux interblocages !

- Interblocage = état non volontaire de blocage définitif entre tâches
 - Provoqué par des attentes « mal réfléchies », soit sur des signaux logiciels, soit sur des sémaphores, soit sur une combinaison des deux à la fois...
- Définition de l'interblocage généralisé :

Un ensemble de tâches est en interblocage si chaque tâche attend la libération d'une ressource allouée à une autre appartenant à l'ensemble.
- Exemple d'interblocage :
 - Une tâche T1 détient une ressource R1 et attend une autre ressource, R2, qui est utilisée par une tâche T2.
 - La tâche T2 détient la ressource R2 et attend la ressource R1.
 - Les deux tâches vont attendre indéfiniment : T1 attend T2, et T2 attend T1



```

main()
{
  creer(Tache_T1);
  creer(Tache_T2);
  Vn(mutexA,1);
  Vn(mutexB,1);
  executer();
}

Tache_T1()
{
  while(1) {
    P(mutexA);
    ... // ressource R1
    P(mutexB);
    ... // ressource R2
    Vn(mutexB,1);
    ...
    V(mutexA);
    ...
  }
}

Tache_T2()
{
  while(1) {
    P(mutexB);
    ... // ressource R2
    P(mutexA);
    ... // ressource R1
    V(mutexA);
    ...
    V(mutexB);
    ...
  }
}

```



Il fallait permuter mutexB avec mutexA dans une des tâches

En résumé, pour provoquer un interblocage entre 2 tâches T1 et T2 qu'avec des sémaphores, il faut...

- par exemple, oublier de conserver le même ordre des P et des V
- ou bien encore, inclure plusieurs exclusions mutuelles de manière désordonnée... (accès croisés aux ressources)

- Avec une double permutation = interblocage !

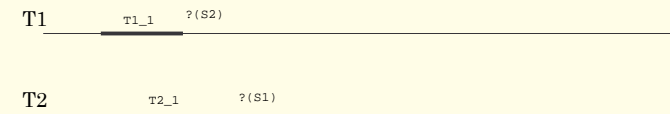
```

main()
{
  creer(Tache_T1);
  creer(Tache_T2);
  executer();
}

Tache_T1()
{
  while(1) {
    // T1_1
    wait(S2);
    signal(S1);
    // T1_2
  }
}

Tache_T2()
{
  while(1) {
    // T2_1
    wait(S1);
    signal(S2);
    // T2_2
  }
}

```



Interblocage évité si « signal » avant « wait »

■ **Prévenir :**

- Rôle du développeur...
- Pour chaque tâche, écrire les P(), les wait et les accès aux ressources toujours dans le **même ordre** ;

■ **Eviter :**

- RTOS alloue les ressources avec précaution.
- Si l'allocation d'une ressource peut conduire à un interblocage, elle est retardée jusqu'à ce qu'il n'y ait plus de risque.

■ **Détecter :**

- RTOS construit et analyse dynamiquement le graphe d'allocation des ressources (graphe des conflits)

■ **Guérir** (reprendre) :

- En Temps Réel par le RTOS
- Peu évident à réaliser
- Retirer temporairement une ressource à une tâche pour l'attribuer à une autre
- Qui revient à terminer une tâche impliquée dans l'interblocage et effectuer les V() manquants pour libérer les autres tâches

■ Principe de l'allocation par le RTOS :

- Déterminer si l'attribution de la ressource est sûre (mène vers un état sûr)
 - Si oui, attribution immédiate
 - Sinon, la ressource n'est pas accordée.
- Etat sûr : toutes les tâches peuvent terminer leurs exécutions
 - Il existe un ordre d'allocation de ressources qui permet à toutes les tâches de se terminer

■ Algorithme de détermination d'un état est sûr, manipulant :

- n tâches : T1, ..., Tn manipulant m types de ressources
- « Existantes » E(m) : vecteur totalisant les ressources
E(j) est le nombre total de ressources de type j
- « Courantes » C(n,m) : matrice des allocations courantes (ressources attribuées)
C(i,j) : nombre de ressources j détenues par la tâche Ti
- « Requêtes » R(n,m) : matrice des requêtes
R(i,j) : nombre de ressources j qui manquent à la tâche Ti
- « Allouables » A(m) : vecteur des ressources qui peuvent être allouées, car disponibles
A(j) : nombre de ressources j disponibles

■ Basé sur un modèle de prêt d'un banquier à ses clients

■ Algorithme (Dijkstra 1965) :

1) Rechercher une tâche Ti qui peut obtenir les ressources demandées :

- tâche Ti non marquée dont la ligne R(i) est inférieure ou égale à A
- toutes les ressources demandées sont disponibles : $\forall j, R(i,j) \leq A(j)$

2) Si cette tâche existe, alors :

- Cette tâche obtiendra les ressources, s'exécutera, et se terminera. Elle libérera alors les ressources déjà détenues, et ne sera pas sujet à un interblocage :
- Marquer la tâche Ti
- Actualiser les ressources disponibles en ajoutant la ligne C(i) à A

Sinon :

- Les tâches non marquées sont en interblocage

3) Si toutes les tâches sont marquées, alors :

- l'état est sûr, donc, fin de l'algorithme

Sinon :

- Retour en 1)

■ Inconvénient : connaître les besoins des tâches en ressources

■ 4 tâches T1, T2, T3 et T4

■ 3 types de ressources : R1, R2 et R3, avec 1 de type R1, 3 de R2 et 4 de R3.

■ Etat initial : E = (1 3 4)

$$\begin{array}{l}
 \text{T1, qui avait une R3,} \\
 \text{en redemande une autre...}
 \end{array}
 \quad
 C = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}
 \quad
 R = \begin{bmatrix} 0 & 0 & 2 \\ 0 & 1 & 2 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{bmatrix}
 \quad
 \begin{array}{l}
 A = E - \text{Total} \\
 A = (1 \ 3 \ 4) - (1 \ 2 \ 3) \\
 A = (0 \ 1 \ 1)
 \end{array}$$

Total : (1 2 3)

■ Etape 1 : marquage T4

$$C = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ - & - & - \end{bmatrix}
 \quad
 R = \begin{bmatrix} 0 & 0 & 2 \\ 0 & 1 & 2 \\ 0 & 2 & 1 \\ - & - & - \end{bmatrix}
 \quad
 \begin{array}{l}
 A = (0 \ 1 \ 1) + (0 \ 1 \ 0) \\
 A = (0 \ 2 \ 1)
 \end{array}$$

■ Etape 2 : marquage T3

$$C = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ - & - & - \\ - & - & - \end{bmatrix}
 \quad
 R = \begin{bmatrix} 0 & 0 & 2 \\ 0 & 1 & 2 \\ - & - & - \\ - & - & - \end{bmatrix}
 \quad
 \begin{array}{l}
 A = (0 \ 2 \ 1) + (0 \ 0 \ 1) \\
 A = (0 \ 2 \ 2)
 \end{array}$$

■ Etape 3 : marquage T2

$$C = \begin{bmatrix} 1 & 0 & 2 \\ - & - & - \\ - & - & - \\ - & - & - \end{bmatrix}
 \quad
 R = \begin{bmatrix} 0 & 0 & 2 \\ - & - & - \\ - & - & - \\ - & - & - \end{bmatrix}
 \quad
 \begin{array}{l}
 A = (0 \ 2 \ 2) + (0 \ 1 \ 0) \\
 A = (0 \ 3 \ 2)
 \end{array}$$

■ Etape 4 : marquage T1

- Fin de l'algorithme, toutes les tâches sont marquées \Rightarrow Etat est sûr

V-4 Exemple 2 d'évitement d'interblocage

- 4 tâches T1, T2, T3 et T4
- 3 types de ressources : R1, R2 et R3, avec 1 de type R1, 3 de R2 et 4 de R3.
- Etat initial :

$$E = \begin{pmatrix} 1 & 3 & 4 \end{pmatrix} \quad C = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad R = \begin{bmatrix} 0 & 0 & 2 \\ 0 & 1 & 2 \\ 0 & \boxed{3} & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$A = E - \text{Total} \quad A = \begin{pmatrix} 1 & 3 & 4 \end{pmatrix} - \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 \end{pmatrix}$$
- Etape 1 : marquage T4

$$C = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ - & - & - \end{bmatrix} \quad R = \begin{bmatrix} 0 & 0 & 2 \\ 0 & 1 & 2 \\ 0 & 3 & 1 \\ - & - & - \end{bmatrix} \quad A = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$
- Etape 2 : aucune tâche restante ne peut s'exécuter :

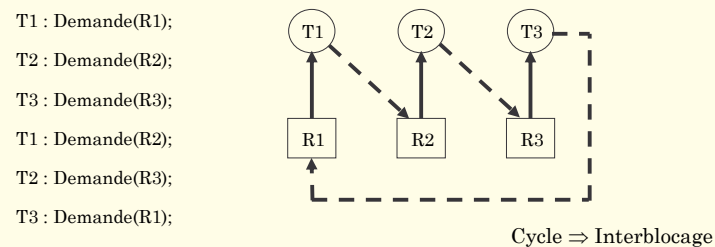
$$\begin{aligned} T1 : (0 \ 0 \ 2) &> (0 \ 2 \ 1) \\ T2 : (0 \ 1 \ 2) &> (0 \ 2 \ 1) \\ T3 : (0 \ 3 \ 1) &> (0 \ 2 \ 1) \end{aligned}$$
- Fin de l'algorithme :
Etat conduisant à un interblocage des trois tâches T1, T2 et T3 sur le partage des ressources R2 et R3.

V-4 Détecter l'interblocage : graphe d'allocation des ressources

- Graphe d'allocation :
 - pour chaque tâche, les ressources détenues et demandées
- Graphe biparti composé de nœuds et d'arcs :
 - Tâches : T_i
 - Ressources : R_i
 - La ressource allouée : \longrightarrow
arc orienté de la ressource vers une tâche
 - La ressource en requête : $---\longrightarrow$
(blocage d'une tâche en attente d'une ressource)
arc orienté de la tâche vers la ressource
- Fréquence de construction du graphe par le RTOS :
 - Périodiquement
 - A chaque modification du graphe suite à une demande d'une ressource coûteuse en termes de temps processeur
 - Lorsque l'utilisation du processeur est inférieure à un certain seuil la détection peut être tardive

V-4 Exemple 1 : graphe détection d'interblocage

- Trois tâches T1, T2 et T3 utilisant trois ressources R1, R2 et R3 ainsi :
 - T1 : Demande(R1); Demande(R2); accès ressources; Libère(R1); Libère(R2);
 - T2 : Demande(R2); Demande(R3); accès ressources; Libère(R2); Libère(R3);
 - T3 : Demande(R3); Demande(R1); accès ressources; Libère(R3); Libère(R1);
 - Si exécution séquentielle (T1 suivie de T2, suivie de T3) : aucun problème !
 - Si exécution entrelacée circulaire (T1_1; T2_1; T3_1; T1_2; ...) : interblocage
- Construction dynamique du graphe :



V-4 Exemple 2 : graphe détection d'interblocage

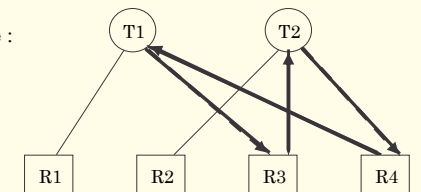
- Deux tâches T1 et T2 utilisant 4 ressources R1, R2, R3 et R4
 - T1 : utilise R1, R3 et R4 pour son traitement
 - T2 : utilise R2, R3 et R4 pour son traitement
- Interblocage potentiel ?
- Si oui, dans quel ordre faire les demandes pour éviter l'interblocage ?
- Construction statique du graphe :

Interblocage si T1 fait :

Demande(R4);
Demande(R3);

Pendant que T2 fait :

Demande(R3);
Demande(R4);



Cycle potentiel \Rightarrow Interblocage potentiel

Dans quel ordre pour éviter l'interblocage ?

Prévenir :

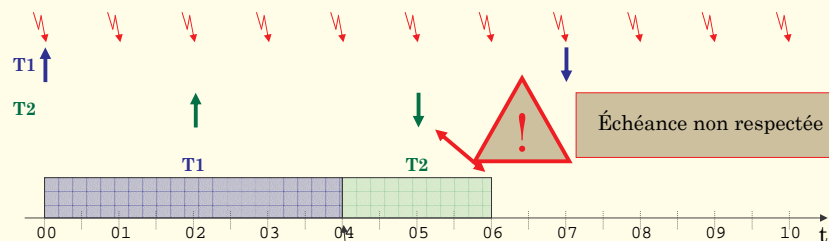
Pour chaque tâche, écrire les accès toujours dans le **même ordre**

- Rôles de l'ordonnanceur :
 - Gérer le processeur, en ligne, et à intervalles réguliers...
 - ... pour gérer les tâches de l'application de l'utilisateur.
 - Il doit permettre aux tâches de respecter leurs contraintes de temps.
 - L'ordonnanceur est généralement appelé dans l'ISR d'une interruption liée à un timer (« Tic » système), pour garantir les intervalles réguliers.
- Principe :
 - Attribuer le processeur à l'une des tâches prêtes (ou en-cours).
 - L'attribution est faite suivant un critère de **priorité** d'exécution.
- Les stratégies d'ordonnancement :
 - Stratégie = politique d'ordonnancement
 - Choisie lors de l'implémentation du système embarqué
 - Fixée une fois pour toute
- Deux grandes familles d'ordonnanceurs
 - Famille en fonction de la stratégie de « préemption » :
 - préempter = autoriser ou non l'arrêt provisoire d'une tâche en exécution
 - Ordonnanceurs **non préemptifs** et ordonnanceurs **préemptifs**

- Ordonnancement **non préemptif**
 - Le processeur est laissé à la tâche courante même si, dans l'intervalle, d'autres tâches sont passées dans l'état prêt, avec une **priorité pourtant plus grande**
 - L'ordonnanceur n'est pas appelé dans l'ISR, mais via une primitive qui change l'état de la tâche courante (bloqué, prêt ou inexistant).
 - Réservé au temps réel mou : contraintes de temps non sévères
 - NB : avantage : aucun problème d'accès aux ressources partagées...
- Ordonnancement **préemptif**
 - La tâche en-cours perd le processeur, si une tâche de priorité plus grande (jugée donc plus urgente) passe dans l'état prêt
 - Ordonnancement **statique** (« à priorité ») :
 - à l'initialisation, chaque tâche se voit attribuer une valeur de priorité, la tâche conserve cette priorité jusqu'à sa destruction.
 - Ordonnancement **dynamique** (« à échéance ») :
 - Les priorités des tâches peuvent évoluer avec le temps qui passe
 - L'échéance, diminuant avec le temps, augmente la priorité dynamique de la tâche
- NB : les deux techniques (statiques et dynamiques) d'ordonnanceurs préemptifs peuvent parfaitement **cohabiter** dans le même système !
 - dans ce cas, dynamiques toujours plus prioritaires que statiques

- Fonctionnement de l'ordonnanceur :
 - Ordonnancement type « premier arrivé, premier servi »,
 - Et non préemptif
- Exemple à deux tâches, modélisées par :
 - C : durée d'exécution (maximale) du code monolithique
 - D : échéance (« Deadline »)
 - S : date de réveil (« Start time »)

Tâche	S	C	D
T1	0	4	7
T2	2	2	5

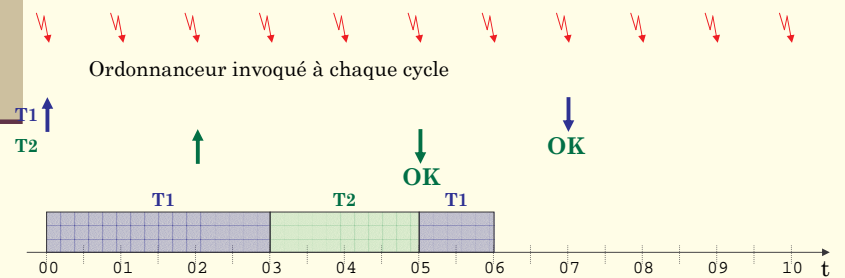


Invocation de l'ordonnanceur

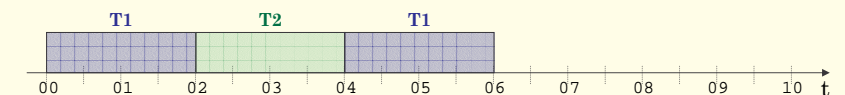
- Même exemple à deux tâches :

Tâche	S	C	D
T1	0	4	7
T2	2	2	5

- On suppose maintenant que la **préemption** est autorisée.
- L'ordonnancement avec respect des échéances est possible :



Il y a même plusieurs solutions possibles !

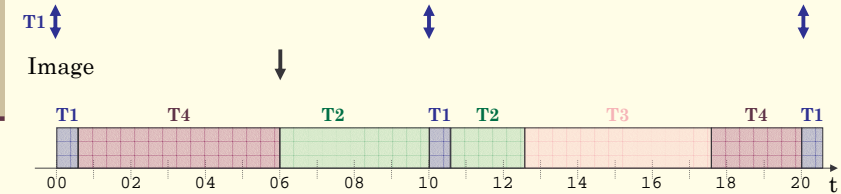


V-5 Exemple 2 d'ordonnanceur préemptif statique à priorités

- Application de contrôle d'une voiture par vision :
 - Une caméra se charge de l'acquisition des images,
 - un ordinateur embarqué réalise le traitement d'images,
 - en fonction des résultats, commande la colonne de direction de la voiture.
 - Les commandes doivent être envoyées à période précise aux actionneurs de la colonne de direction (par exemple une commande toutes les 10 ms).
- Description « formelle » des tâches :
 - C : durée d'exécution (maximale) du code monolithique
 - Prio : priorité
 - P : période, si la tâche est périodique
- Analyse de l'application, et découpage en tâches :
 - Tâche T1 de commande aux actionneurs : $C(T1) = 0.5\text{ms}$, $P(T1) = 10\text{ms}$
 - Tâche T2 d'acquisition exécutée lors de l'arrivée d'une image : $C(T2) = 6\text{ms}$
 - Tâche T3 de traitement de l'image (toujours après T2) : $C(T3) = 5\text{ms}$
 - Tâche T4 d'affichage de l'image, non critique : $C(T4) = 30\text{ms}$
- Choix judicieux des priorités :
 - $\text{Prio}(T1) = 0$ (priorité la plus forte)
 - $\text{Prio}(T2) = 1$
 - $\text{Prio}(T3) = 2$
 - $\text{Prio}(T4) = 3$ (priorité la plus faible)

V-5 Exemple 2 d'ordonnanceur préemptif statique à priorités

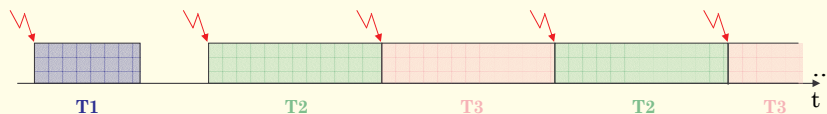
Tâche	P	C	Prio
T1	10	0.5	0
T2	Invoquée sur Image	6	1
T3	Invoquée par T2	5	2
T4	Invoquée par T3	30	3



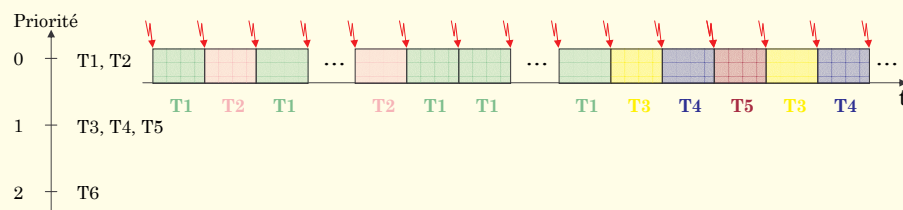
- Analyse du déterminisme :
 - T1 s'exécute toujours en 0.5 ms, toutes les 10ms
 - T2 est traitée au mieux en 6 ms (T2) et au pire en 6.5 ms (T2+T1) ;
 - T3 est traitée au mieux en 5 ms (T3) et au pire en 11,5 ms (T3+T2+T1).
- Si fréquence d'arrivée d'images correcte : elle ne peut être interrompue par T2
 Le cas le pire est alors 5.5 ms (T3 + T1) ;
- T4 n'aura peut-être pas le temps de s'exécuter entièrement à chaque arrivée d'image, donc, certaines images ne seront pas affichées...

V-5 Exemple 3 d'ordonnanceur préemptif statique : Tourniquet

- Politique préemptif statique à tourniquet (« Balayage cyclique ») :
 - ordonnancement très généraliste, « efficace » quelque soit le type des tâches à ordonnancer.
- Deux versions de tourniquet :
 - Ordonnanceur préemptif à tourniquet simple (« Round Robin »)



- Politique à niveaux : Ordonnanceur préemptif à tourniquet à priorités

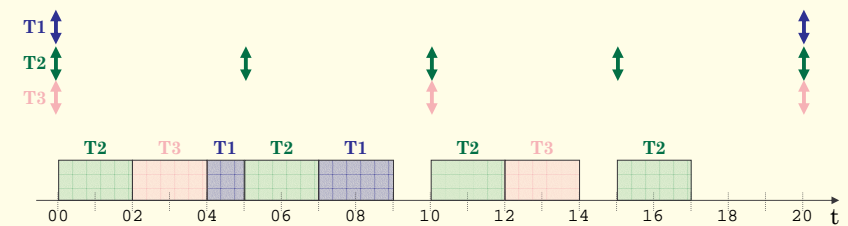


V-5 Exemple 4 d'ordonnanceur préemptif statique : « Rate Monotonic »

- « Rate Monotonic », défini par Liu et Layland en 1973
- Réservée aux tâches périodiques, sous échéances, modélisées par :
 - C : durée d'exécution (maximale) du code monolithique
 - D : échéance (« Deadline ») = P : période
- Priorité inversement proportionnelle à la période $\text{Prio}(Ti) = \frac{1}{P_i}$
- Exemple à 3 tâches T1, T2 et T3 :

Tâche	P	C
T1	20	3
T2	5	2
T3	10	2

Au départ :
 $\text{Prio}(T2) > \text{Prio}(T3) > \text{Prio}(T1)$



Exécution cyclique (PPCM des périodes)

- Tâches périodiques ou non, sous échéances, modélisées par :
 - C : durée d'exécution (maximale) du code monolithique
 - D : échéance (« Deadline »)
 - S : date de réveil (« Start time »)
 - F : date de fin (« Finish time »)
 - P : période (« Period »), si tâche périodique
- Condition d'ordonnabilité :

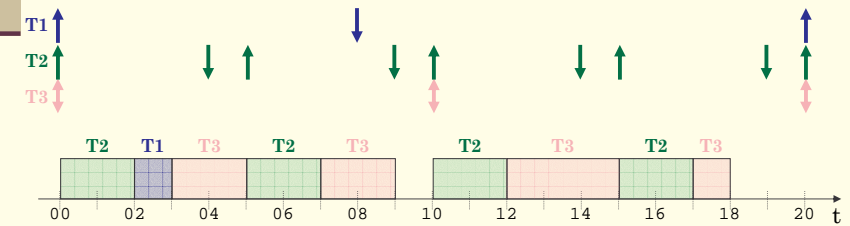
$$\sum_i \frac{C_i}{P_i} < 1$$

- E.D.F : « Earliest Deadline First »

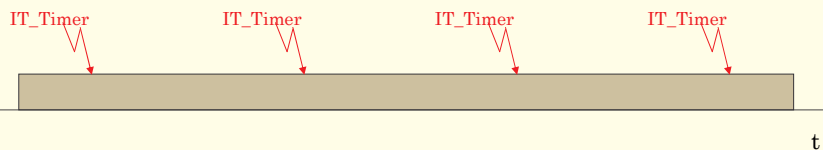
La priorité la plus grande est donnée à la tâche dont l'échéance absolue est la plus proche.

- Exemple d'ordonnement « Earliest Deadline First » à 3 tâches :

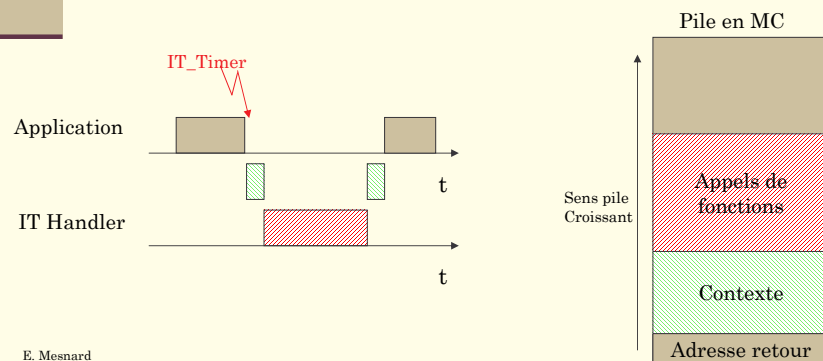
Tâche	P	C	D
T1	20	1	8
T2	5	2	4
T3	10	4	10



Installation d'un Timer : période T = « Tic »



Installation de l'IT associée (gestionnaire de l'IT)



Implémentation sur DSP Texas Instruments C31

```

/* ===== */ /* Configurations des parametres Timer et pile */
/* ----- */
author
input
output
called by
called to
function
*****
void start_kernel()
BEGIN
/* Choix de la prochaine tache a executer (resultat dans newtskid) */
schedule();
curtskid = newtskid;
#ifdef C31_MODE
/* installation, initialisation et demarrage du timer */
{
volatile unsigned int* TimerGlobalControl = (volatile unsigned int*) 0x808020;
volatile unsigned int* TimerCounter = (volatile unsigned int*) 0x808024;
volatile unsigned int* TimerPeriod = (volatile unsigned int*) 0x808028;

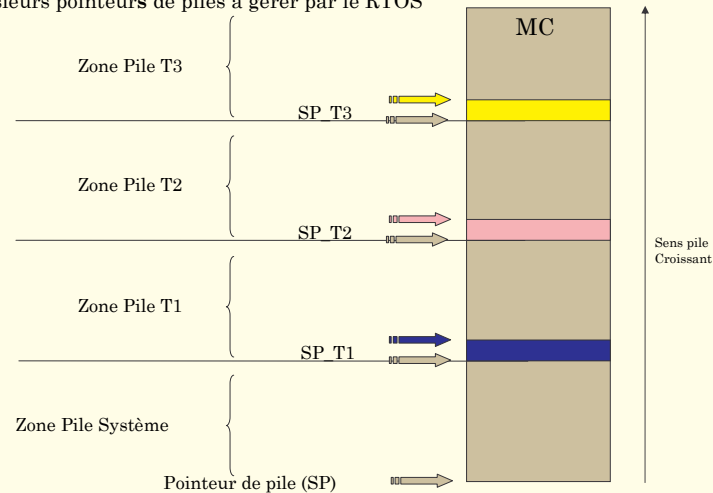
*TimerPeriod = Timer0_Period;
*TimerGlobalControl = Timer0_GlobalControl;
}
#endif

/* Restauration eventuelle (pour les sporadiques) de contexte */
restore_context();
END /* de start_kernel */

```


Création de contextes pour chaque tâche

⇒ Plusieurs pointeurs de piles à gérer par le RTOS



Pointeur « Frame » : bases pour chaque tâche (SP_Ti_init)

Pointeur « Stack » : courant pour chaque tâche (SP)

```

/*****
 * MESNARD Emmanuel                               ISIMA-LIMOS
 * Octobre 2006
 *
 * Definitions des structures
 * NOSTRUCT.H version 1.4
 *****/

#ifndef _NOSTRUCT_
#define _NOSTRUCT_

struct Task {
    /* Structure des Taches */
    int tsk_adr; /* adresse de la tache */
    int id; /* identificateur de la tache */
    int st_ini_offset; /* pointeur initial pile (SpTsk) ou offset (Per) */
    int st_cur_prd; /* pointeur courant pile (Sp) ou periode (Per) */
    int evt; /* evenement dont cette tache est en attente */
    int pri; /* priorite */
    int tsk_state; /* etat de la tache */
    struct Task *nxtskptr; /* pointeur de file sur tache suivante */
};

struct Message {
    /* Structure des Messages */
    int msg_sem; /* semaphore d'operations 'nreceive' autorisees */
    int *data; /* pointeur sur les donnees */
};

struct Event {
    /* Structure des Evenements */
    int evt_sem; /* semaphore d'operations 'nwait' autorisees */
    int tick_counter; /* horloge interne de l'evenement */
};

```

```

void save_context()
BEGIN
    /* declaration des variables globales utilisees */
    asm(" .global _kernel_frame_ptr");
    asm(" .global _kernel_stack_ptr");
    asm(" .global _frame_ptr");
    asm(" .global _return_address");
    asm(" .global _vrai_retour");

    /* on regarde ou il faudra revenir a la fin de save_context */
    asm(" LDIU *-AR3(1), RO");
    asm(" STI RO, @_vrai_retour");

    /* On travaille apres la pile utilisateur */
    asm(" LDIU *AR3, AR3"); /* on ne tient plus compte de l'appel de save_context */
    asm(" LDIU *AR3, AR3"); /* ni de l'appel a nreceive, ou nwait */
    asm(" SUBI 6,SP"); /* il y a eu 6 push, a oublier */
    /* remise a jour du pointeur de debut de frame de pile pour cette tache */
    asm(" STI AR3, @_frame_ptr");

    /* On sauve les registres dans cette pile */
    /* ***** */

    /* On reserve une place dans la pile pour l'adresse de retour */
    asm(" PUSH RO"); /* c'est un PUSH bidon, ecrase a la fin de cette fonction */

    /**** Sauvegarde des registres ****/
    asm(" PUSH ST"); /* etat */
    asm(" PUSH AR3");

    asm(" PUSH RO");
    asm(" PUSHF RO");
    asm(" PUSH R1");

```

```

void restore_context()
BEGIN
    asm(" .global _frame_ptr");
    asm(" .global _stack_ptr");
    asm(" .global _return_address");

    if (Tsk_Tbl[curtskid].pri != MAXPRI+1) {
        /* Restaure le contexte si la nouvelle tache est sporadique */
        frame_ptr = Tsk_Tbl[curtskid].st_ini_offset;
        stack_ptr = Tsk_Tbl[curtskid].st_cur_prd;

        /* On travaille avec la nouvelle pile */
        asm(" LDIU @_frame_ptr, AR3");
        asm(" LDIU @_stack_ptr, SP");

        /* On recupere les registres */
        asm(" *** Recuperation des registres ***");

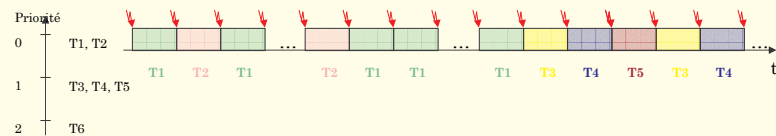
        /* Registres supplementaires */
        asm(" POP IOF");
        asm(" POP IF");
        asm(" POP IE"); /* Interruptions */
        asm(" POP DF"); /* data-page */
        asm(" POP AR7");
        asm(" POP AR6");

        /* registres geres par gestion d'IT */
        asm(" POP BK"); /* taille du bloc */

        asm(" POP AR5");
        asm(" POP AR4");
    }

```

Par exemple : Tourniquet à priorités, en multi-niveaux



Présence d'une tâche de fond...

```

/***** kernel_tsk() *****/
author      : MESNARD Emmanuel
input       : No
output      : No
called by   : init_kernel()
called to   : No
function    : Tache de fond necessaire au noyau
*****/
#ifdef CS1_MODE
int kernel_tsk()
BEGIN
    while(1) {
        /* TODO :
        /* Mettre ici le code de monitoring du systeme */
        /* pour traitement de la supervision */
    };
END /* de kernel_tsk */

```

Ordonnanceur dans (ou appelé via) l'ISR

```

void schedule()
BEGIN
    struct Task *courante;
    int priorite;

    if ( (curtskid != NO_TSK_ID) && (Tsk_Tbl[curtskid].pri == MAXPRIO+1) ) {
        /* on garde toujours la periodique en cours d'execution */
        /* on garantit ainsi qu'une periodique ne preempt pas une autre */
        newtskid = curtskid;
    } else {
        /* soit il n'y a plus de tache en execution, soit c'etait une sporadique */

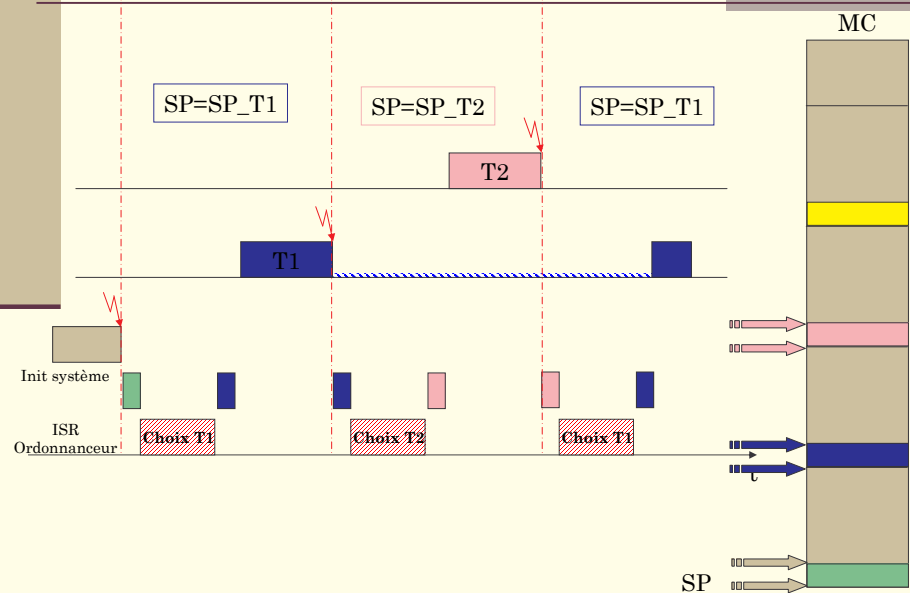
        /* on remet la tache sporadique en fin de liste */
        if (curtskid != NO_TSK_ID) {
            insert_tsk_lst(curtskid, &Ready_lst[Tsk_Tbl[curtskid].pri]);
        }

        /* on reprend une nouvelle tache, la plus prioritaire ! */
        courante = NULL;
        for (priorite=MAXPRIO+1; (courante = Ready_lst[priorite]) == NULL; priorite--);

        /* La tache a executer est la tete de liste */
        newtskid = courante->id;

        /* retrait de la tete de liste dans le cas d'une sporadique */
        if (priorite != MAXPRIO+1) {
            Ready_lst[priorite] = courante->nxtskptr;
        }
    }
END /* de schedule */

```



```

void c_int09()
BEGIN
    int i;
    struct Task *curtskptr;

    /* Registres supplementaires */
    asm(" PUSH AR6");
    asm(" PUSH AR7");
    asm(" PUSH DP"); /* data-page */
    asm(" PUSH IE"); /* Interruptions */
    asm(" PUSH IF");
    asm(" PUSH IOP");
    Disable_IT;

    /* Avancement du temps sur les evenements */
    for (i = 0; i < MAXEVT; i++) {
        Evt_Box[i].tick_counter--;
        if (Evt_Box[i].tick_counter < 0) {
            /* on vient de decrementer un compteur deja a 0
            /* soit deja traite auparavant, soit un NOWATCHDOG ou INFINITE */
            Evt_Box[i].tick_counter = 0;
        } else if (Evt_Box[i].tick_counter == 0) {
            /* il faut reveiller toutes les taches en attente de la cle i */
            while (Evt_Suspended_lst[i] != NULL) {
                curtskptr = Evt_Suspended_lst[i];
                Evt_Suspended_lst[i] = curtskptr->nxtskptr; /* on ote la tete de liste */
                curtskptr->tsk_state = READY_SYS; /* que l'on re-active */
                insert_tsk_lst(curtskptr->id, &Ready_lst[curtskptr->pri]);
            } /* du while */
            Evt_Box[i].evt_sem = 0; /* personne n'attend plus ce semaphore */
        }
    } /* du for */

    /* Choix de la prochaine tache a executer (resultat dans newtskid) */
    schedule();

```

```

if (curtskid=newtskid) { /* Preemption => changement de contexte */
    /* Mise a jour des pointeurs de pile */
    asm(" STI SP, @_stack_ptr");
    asm(" STI AR3, @_frame_ptr");

    /* On memorise pour cette tache les nouveaux pointeurs */
    Tsk_Tbl[curtskid].st_cur_prd = stack_ptr;
    Tsk_Tbl[curtskid].st_ini_offset = frame_ptr;

    /* On traite maintenant la nouvelle tache a executer */
    curtskid = newtskid;
    stack_ptr = Tsk_Tbl[curtskid].st_cur_prd;
    frame_ptr = Tsk_Tbl[curtskid].st_ini_offset;

    /* On travaille avec la nouvelle pile */
    asm(" LDIU @_frame_ptr, AR3");
    asm(" LDIU @_stack_ptr, SP");

} else { /* Pas de preemption */
    /* On re-autorise les IT, puis on repart dans la tache en cours */
    Enable_IT;
}

/* Registres supplementaires */
asm(" POP IOF");
asm(" POP IF");
asm(" POP IE"); /* Interruptions */
asm(" POP DF"); /* data-page */
asm(" POP AR7");
asm(" POP AR6");

/* Pas d'Enable_IT qui se fera dans restore_context qui suit... */
END /* de c_int09 */

```

Choix d'une implémentation pour la synchronisation (échange d'événements)

Par exemple : tout événement de synchronisation est un **sémaphore**

Présence d'un compteur d'événements

1 émission \Rightarrow 1 autorisation de lecture (compteur ++)

1 réception \Rightarrow 1 consommation de lecture (compteur --)

Attente bloquante si compteur devient négatif

Implémentation basique :

P(Evt) = **wait**(Evt)

V(Evt) = **signal**(Evt, 1)

Vn(Evt, Evt_Counter) = **signal**(Evt, Evt_Counter)

Implémentation avec chien de garde :

P(Evt) = **wait**(Evt, NOWATCHDOG)

P(Evt, Watchdog) = **wait**(Evt, Watchdog)

Sleep(Evt, duration) = **wait**(Evt, duration)

V(Evt) = **signal**(Evt, 1)

Vn(Evt, Evt_Counter) = **signal**(Evt, Evt_Counter)

```

void nsignal(int evt, int evt_counter)
BEGIN
    struct Task *curtskptr;
    int i;

    Disable_IT;

    DEBUG(evt_counter = ((evt_counter < 0) ? 0 : evt_counter))
    for(i=0; i<evt_counter; i++) {
        Evt_Box[evt].evt_sem++;
        if (Evt_Box[evt].evt_sem < 1) {
            /* ce nsignal reveille la premiere tache de la liste des suspendues */
            curtskptr = Evt_Suspended_lst[evt];
            Evt_Suspended_lst[evt] = curtskptr->nxtskptr; /* on ote la tete de liste */
            curtskptr->tsk_state = READY_USR; /* que l'on re-active */
            insert_tsk_lst(curtskptr->id, &Ready_lst[curtskptr->pri]);

            if (Evt_Box[evt].evt_sem == 0) {
                /* ce nsignal vient de debloquer la derniere tache bloquee */
                Evt_Box[evt].tick_counter = 0;
            } /* du if (... < 1) */
        } /* du for */
    }

    Enable_IT;
END /* de nsignal */

```

```

void nwait(int evt, int duration)
BEGIN
    asm(" .global _return_address");
    Disable_IT;
    if (Tsk_Tbl[curtskid].pri == MAXPRIO+1) { /* tache periodique */
        DEBUG(print("Erreur : nwait %d sur une tache periodique.\n", evt));
    } else { /* tache sporadique */
        Evt_Box[evt].evt_sem--; /* on decremente le semaphore */
        if (Evt_Box[evt].evt_sem < 0) {
            /* il n'y a pas eu de nsignal => ce nwait est bloquant */
            Evt_Box[evt].tick_counter = duration; /* remise a jour de la duree */
            /* on suspend la tache courante */
            Tsk_Tbl[curtskid].tsk_state = SUSPEND;
            Tsk_Tbl[curtskid].evt = evt; /* suspension sur evenement evt */
            insert_tsk_lst(curtskid, &Evt_Suspended_lst[evt]);
        }
    }

#ifdef C31_MODE
    /* memorisation de l'instant d'appel de ce nwait */
    asm(" LDIU *-AR3(1), RO");
    asm(" STI RO, @_return_address");
    /* le retour devra se faire juste apres le nwait */
    return_address += 1;
    /* Sauve le contexte de la tache sporadique preemptee */
    frame_ptr = Tsk_Tbl[curtskid].st_ini_offset; /* On re-memorise le frame */
#endif

    save_context();
    /* on repart en ordonnancement sans tache en cours d'execution */
    curtskid = NO_TSK_ID;
    /* Choix de la prochaine tache a executer (resultat dans newtskid) */
    schedule();
    curtskid = newtskid;
    /* Restauration eventuelle de contexte */
    restore_context();
}

/* definition des synonymes des fonctions */
#define nsleep      nwait
#define V(sem)      nsignal(sem, 1)
#define Vn(sem,nbre) nsignal(sem,nbre)
#define P(sem)      nwait(sem, NOWATCHDOG)

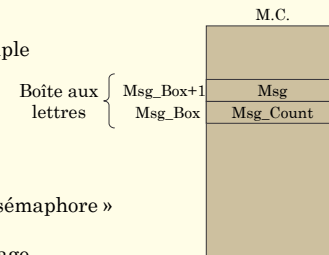
```

Choix d'une implémentation pour la synchronisation (échange d'événements)

Communication indirecte, via la mémoire partagée

Par exemple :

Boîte aux lettres avec mémorisation simple



Communication type « synchronisation sémaphore »

Présence d'un entier de comptage
= nombre de fois que le message peut être lu

Emission **non bloquante**

Réception sur la boîte aux lettres :

- **bloquante** si le message n'est pas encore arrivé,
- **non bloquante** sinon.

```

/***** nsend() *****/
author      : MESNARD Emmanuel
input       : msg, msg_counter, *data, value_size
output      : No
called by   : User application
called to   : insert_tsk_lst()
function    : Emission d'un message
*****/

void nsend(int msg, int msg_counter, int *data, int value_size)
BEGIN
    struct Task *curtskptr;
    int i;

    Disable_IT;

    /* Mise a jour des nouvelles donnees recues */
    Msg_Box[msg].data = data;

    DEBUG(msg_counter = ((msg_counter<0) ? 0 : msg_counter))

    for(i=0; i<msg_counter; i++) {
        Msg_Box[msg].msg_sem++;
        if (Msg_Box[msg].msg_sem < 1) {
            /* ce nsend reveille la premiere tache de la liste des suspendues */
            curtskptr = Msg_Suspended_lst[msg];
            Msg_Suspended_lst[msg] = curtskptr->nxtskptr; /* on ote la tete de liste */
            curtskptr->tsk_state = READY_USR; /* que l'on re-active */
            insert_tsk_lst(curtskptr->id, &Ready_lst[curtskptr->pril]);
        } /* du if (... < 1) */
    } /* du for */

    Enable_IT;
END /* de nsend */

```

```

void nreceive(int msg, int **data)
BEGIN
    asm(" .global _return_adress");

    Disable_IT;

    Msg_Box[msg].msg_sem--; /* on decremente le semaphore */

    if (Msg_Box[msg].msg_sem >= 0) {
        /* il y a deja eu un nsend => ce nreceive n'est pas bloquant */
        *data = Msg_Box[msg].data;
    } else {
        /* ce nreceive doit etre bloquant... que pour les taches sporadiques */

        if (Tsk_Tbl[curtskid].pri == MAXPRIO+1) { /* tache periodique */
            DEBUG(print("Erreur : nreceive %d sur une tache periodique.\n", msg));
            /* on retourne quand meme l'ancienne de valeur de la donnee */
            Msg_Box[msg].msg_sem++; /* on re-incremente le semaphore */
            *data = Msg_Box[msg].data;
        } else { /* tache sporadique */
            /* on suspend la tache courante */
            Tsk_Tbl[curtskid].tsk_state = SUSPEND;
            Tsk_Tbl[curtskid].evt = msg; /* suspension sur message msg */
            insert_tsk_lst(curtskid, &Msg_Suspended_lst[msg]);
        }
    }
}

```

```

} else { /* tache sporadique */
    /* on suspend la tache courante */
    Tsk_Tbl[curtskid].tsk_state = SUSPEND;
    Tsk_Tbl[curtskid].evt = msg; /* suspension sur message msg */
    insert_tsk_lst(curtskid, &Msg_Suspended_lst[msg]);
}

#ifdef C31_MODE
/* memorisation de l'instant d'appel de ce nreceive */
asm(" LDIU *-AR3(1), RO");
asm(" STI RO, @_return_adress");
/* le retour devra se faire juste apres le nreceive */
return_adress += 1;
/* Sauve le contexte de la tache sporadique preemptee */
frame_ptr = Tsk_Tbl[curtskid].st_ini_offset; /* On re-memorise le frame */
#endif

save_context();
/* on repart en ordonnancement sans tache en cours d'execution */
curtskid = NO_TSK_ID;
/* Choix de la prochaine tache a executer (resultat dans newtskid) */
schedule();
curtskid = newtskid;
/* Restauration eventuelle de contexte */
restore_context();
}

Enable_IT;
END /* de nreceive */

```

V-6 Réalisation d'un RTOS : Etape 7 – Programme principal

```

/* *****
/* MESNARD Emmanuel
/* ISIMA-LIMOS
/* Octobre 2006
/*
/*
/* Exemple des philosophes mangeurs de riz
/*
/* Nanos.C version 1.4
/* *****

/* Fichiers d'inclusions systemes
/*
/* -----
/*

#include "nodirect.h"
#include "noconst.h"
#include "nostruct.h"
#include "novar.h"
#include "nokernel.h"

/*****
/* Parametres de l
*****/

/* inclusion des parametres d
#include "nanos.h"

/* identificateurs des evenem
#define chopstick1 0
#define chopstick2 1
#define chopstick3 2
#define room_ticket 3
#define tempo1 4
#define tempo2 5
#define tempo3 6

/* Prototypes des taches de l'utilisateur */
void philosophe1();
void philosophe2();
void philosophe3();

main()
{
    /* Initialisation du noyau */
    init_kernel();

    /* Creation des taches SPORADIQUES de l'application */
    create_sp_tsk((int)philosophe1, 1);
    create_sp_tsk((int)philosophe2, 1);
    create_sp_tsk((int)philosophe3, 1);

    /* Initialisation des semaphores */
    V(chopstick1); /* une baguette entre chaque assiette */
    V(chopstick2);
    V(chopstick3);
    /* On met une autorisation de moins que le nombre de philosophes
    | (ici : room_ticket=3-1=2) */
    Vn(room_ticket,2);

    /* Demarrage du noyau pour finir... */
    start_kernel();
}

```

V-7 RTOS libre : MicoC/OS



« MicroC/OS-II, The Real-Time Kernel »
J. Labrosse, Mc Graw Hill, ISBN : 1-57820-103-9, 70 €

µC/OS est un exécutif temps réel destiné à des environnements de très petite taille construits autour de microcontrôleurs de type Freescale 68HC11. Il est maintenant disponible sur un grand nombre de processeurs et peut intégrer des protocoles standards comme TCP/IP (µC/IP) pour assurer une connectivité IP sur une liaison série par PPP.



<http://www.micrium.com/products/rtos/kernel/rtos.html>

Micrium μ C/OS-II is written mainly in the C programming language. It's a :

portable,
ROMable,
scalable,
preemptive real-time,
deterministic,

multitasking kernel for microprocessors, microcontrollers and DSPs.

µC/OS-II can manage up to 254 application tasks and provides the following services: Semaphores, Event Flags, Mutual Exclusion Semaphores (to reduce priority inversions), Message Mailboxes, Message Queues, Task Management (Create, Delete, Change Priority, Suspend/Resume etc.), Time Management, Fixed Sized Memory and Block Management.

Portages : Altera, Analog Devices, ARM, Atmel, Freescale/Motorola, Fujitsu, IBM, Intel, Lattice, Microchip, Mitsubishi, NEC, Renesas/Hitachi, Texas Instruments, Xilinx,

V-7 RTOS libre : Free RTOS



<http://www.freertos.org/>

FreeRTOS is a portable, open source, mini Real Time Kernel - a free to download and royalty free RTOS that can be used in commercial applications.

FreeRTOS is a real-time operating system for embedded devices, being ported to several microcontrollers. It is distributed under a modified version of the GPL. The modification permits users' proprietary code to remain closed source while maintaining the kernel itself as open source, thereby facilitating the use of FreeRTOS in commercial applications.

The FreeRTOS scheduler is designed to be small and simple. It can be configured for both preemptive or cooperative operation. To make the code readable, easy to port, and maintainable, it is written mostly in C, but there are a few assembler functions included where needed.

Portages : ARM, Atmel, Xilinx, Renesas/Hitachi, Microchip, Intel, Freescale/Motorola, Texas Instruments, ...

V-7 RTOS libre : eCos



<http://ecos.sourceforge.org/>

eCos a initialement été développé par la société Cygnus. Il a été racheté ensuite par Red Hat, et est désormais un logiciel libre développé de manière communautaire.

eCos (embedded Configurable operating system) is an open source, royalty-free, real-time operating system intended for embedded systems and applications which need only one process with multiple threads. It can be customized to precise application requirements, with hundreds of options, delivering the best possible run-time performance and minimized hardware needs.

It is programmed in the C programming language, and has compatibility layers and APIs for POSIX and μ TRON. Support for a wide variety of devices including many serial devices, ethernet controllers and FLASH memories. There is also support for PCMCIA, USB and PCI interconnects. A fully featured TCP/IP stack implementing IP, IPv6, ICMP, UDP and TCP over ethernet. Support for SNMP, HTTP, TFTP and FTP are also present.



Portages : Agilent, Altera, ARM, Atmel, Bright Star, Freescale/Motorola, Fujitsu, HP, IBM, Intel, Matsushita, Mitsubishi, NEC, Renesas/Hitachi, Samsung, Sun, Texas Instruments, XScale

ISIMA V-7 RTOS « libre » : Linux embarqués

GRATUIT

Portails :

-  linuxDevices.com
-  <http://uuu.enseirb.fr/~kadionik/embedded/embeddedlinux.html>

Versions embarquées :


MontaVista, BlueCat Linux, Debian, RedHat Embedded, LinuxRouterProject, ELKS, Tom'sRTBT, Small Linux, PeeWee Linux, µCLinux, RTLinux, RTAI, TimeSys, miniLin, Monkey, Tiny Linux, Embedix...



E. Mesnard 81

ISIMA V-7 RTOS « presque » libre : QNX

GRATUIT

 <http://www.qnx.com/developers/qnx4/>

QNX is a commercial POSIX-compliant Unix-like real-time operating system, aimed primarily at the embedded systems market.

Comme tout système utilisant un **micro-noyau**, QNX est basé sur l'idée originale de faire fonctionner une grande partie du système d'exploitation comme un ensemble de petites tâches connues sous le nom de **serveurs**.

Le noyau diffère de ceux plus traditionnels et monolithiques, dans lequel le système d'exploitation est un seul gros programme avec des facultés particulières. L'utilisation d'un micro-noyau permet aux développeurs de désactiver les fonctionnalités qui ne leur sont pas nécessaires, sans avoir à changer de système d'exploitation.



Il est également considéré à la fois comme léger, robuste (micro-noyau), très rapide et complet. Grâce à sa compatibilité POSIX, de nombreuses applications open source(UNIX) ont été portées sur QNX.

QNX est une solution très intéressante car il est, depuis 2007, gratuit pour un usage non commercial.

Particularité : Les outils de développement peuvent être implantés directement sur la cible.

E. Mesnard 82

ISIMA V-7 RTOS commercial : LynxOS

  <http://www.linuxworks.com/rtos/>

An RTOS with Open APIs and Linux ABI compatibility
Full POSIX® conformance in an embedded RTOS
Mission-critical RTOS performance and reliability
Advanced networking feature sets
Latest RTOS technologies for Internet communications

LynxOS components are designed for absolute determinism (hard real-time performance), which means that they respond within a known period of time.
Predictable response times are ensured even in the presence of heavy I/O due to the kernel's unique threading model, which enables interrupt routines to be extremely short and fast.

Portages : Motorola 68010, Intel 80386, ARM, PowerPC, ...

E. Mesnard 83

ISIMA V-7 RTOS commercial : VxWorks

VxWorks 6.x WIND RIVER

http://windriver.com/products/run-time_technologies/Real-Time_Operating_Systems/VxWorks_6x/

Like most RTOSes, VxWorks includes a multitasking kernel with pre-emptive scheduling and fast interrupt response, extensive inter-process communications and synchronization facilities, and a file system.

Major distinguishing features of VxWorks include efficient POSIX-compliant memory management, multiprocessor facilities, a shell for user interface, symbolic and source level debugging capabilities, and performance monitoring.

In addition to the capacity for preemptive scheduling, high reliability, and stringent performance, real-time operating systems offer additional benefits :

- Small memory footprint,
- Fast boot time,
- Low power consumption,
- Long battery life,
- Low cost, Scalability.

Particularité : Cet OS de la firme Wind River, a été employé par la NASA pour les missions spatiales Mars Pathfinder, Stardust, ainsi que pour les deux rovers martiens Spirit et Opportunity. La sonde martienne Mars Reconnaissance Orbiter l'utilise également.

Portages : ARM, Renesas/Hitachi, Microchip, Intel, Freescale/Motorola, Texas Instruments, ...

E. Mesnard 84



Windows CE (also known officially as Windows Embedded CE since version 6.0) is a variation of Microsoft's Windows operating system for minimalistic computers and embedded systems.

Windows CE is a distinctly different kernel (XP Embedded is a NT based system).

Many platforms have been based on the core Windows CE operating system, including Microsoft's AutoPC, Pocket PC 2000, Pocket PC 2002, Windows Mobile 2003, Windows Mobile 2003 SE, Windows Mobile 5.0, Windows Mobile 6 Smartphone 2002, Smartphone 2003 and many industrial devices and embedded systems.

Windows CE even powered select games for the Sega Dreamcast, was the operating system of the controversial Gizmondo handheld, and can partially run on modified Microsoft Xbox game consoles.

Difficilement utilisable pour un petit système embarqué...

Portages : Palm, Pocket PC, tablet PC, game consoles, phones

Conclusion

Conclusion

- La complexité en croissance exponentielle des applications embarquées (source : CEO de Wind River Ken Klein - 2005):
 - 100 000 lignes de code en moyenne en 2003, un million en 2005.
 - On attend un doublement de la taille des applications tous les deux ans. Les demandes en connectivité et sécurité des échanges sont à l'origine de cette explosion.
- Les puissances de calcul et des capacités de stockage disponibles
- L'émergence de solutions à base de circuits logiques programmables (FPGA) pour les applications temps réel critiques.
- $1+1+1=1$!
Le développeur, l'électronicien, et l'expert réseaux ne devront faire plus qu'un. . .

et ça sera un étudiant ISIMA !