

# Temps Réel

Jérôme Pouiller <[j.pouiller@sysmic.org](mailto:j.pouiller@sysmic.org)>



Septembre 2011

- Problématique
- Le monotâche
- Le multitâches
- L'ordonnement
- Le partage de ressources
- Problématiques des OS temps réels

# Qu'est-ce que le temps réel ?

Deux exemples :

- Un système de navigation calculant le meilleur parcours d'un navire.
- Un système de navigation calculant la position d'un navire en mouvement.
- Dans le premier cas le temps nécessaire à l'obtention du calcul est accessoire.
- Dans le deuxième cas, si le calcul est trop long, la position est erronée.

# Le système temps réel et son environnement

**Système** : ensemble d'"activités" correspondant à un ou plusieurs traitements effectués en séquence ou en concurrence et qui communiquent éventuellement entre eux.

Un système temps réel interagit avec son environnement

- Capteurs : signaux et mesure de signaux
- Unité de traitement
- Actionneurs : actions sur l'environnement à des moment précis

**Système temps réel** : un système dont le comportement dépend, non seulement de l'exactitude des traitements effectués, mais également du temps où les résultats de ces traitements sont produits.

En d'autres termes, un retard est considéré comme une erreur qui peut entraîner de graves conséquences.

# Echéances et systèmes temps réel

On distingue différents types d'échéances

- **Echéance dure** : un retard provoque une exception (traitement d'erreur) (exemple : carte son)
- **Echéance molle ou lâche** : un retard ne provoque pas d'exception (exemple : IHM)

On distingue par conséquent différents types de systèmes temps réels

- **Temps réel dur** : les échéances ne doivent en aucun cas être dépassées
- **Temps réel lâche ou mou** : le dépassement occasionnel des échéances ne met pas le système en péril

En plus de la tolérance à l'erreur, nous devons prendre en compte la criticité de l'erreur :

- Téléphone portable
- Carte son professionnelle
- Système de commande d'un robot industriel
- Système de commande d'un avion de ligne

# Prévisibilité, déterminisme, fiabilité

Systèmes temps réels : prévus pour le pire cas.

**Prévisibilité** : Pouvoir déterminer à l'avance si un système va pouvoir respecter ses contraintes temporelles. Connaissance des paramètres liés aux calculs.

**Déterminisme** : Enlever toute incertitude sur le comportement des tâches individuelles et sur leur comportement lorsqu'elles sont mises ensemble.

- variation des durées d'exécution des tâches
- durée des E/S, temps de réaction
- réaction aux interruptions, etc.

**Fiabilité** : comportement et tolérance aux fautes.

Par conséquent, le temps réel possède plusieurs facettes :

- Ingénierie informatique : Algorithmique, développement
- Ingénierie électromécanique : Maitrise de l'environnement physique
- Processus : Maitrise de la qualité du produit, garantie sur les bugs
- Administrative : Certification

**Exemple** : Un airbag est un système temps réel très dur avec une échéance de temps très faible. La puissance du système n'est pas dans sa conception, mais dans sa garantie de qualité. C'est très facile de faire un airbag, c'est beaucoup plus complexe de le garantir.

# Exemple - Combiné GSM

Le système doit combiner :

- Couche physique : émission, réception, activation du vocodeur, mesures du niveau de réception, etc.
- Procédures logicielles : communication avec les base de données du système pour les mises à jour de localisation, transmission des résultats de mesure de qualité, scrutation de messages d'appel, etc.

Les contraintes de temps : 577  $\mu$ s de parole emis puis recus toutes les 4,6 ms.

Les contraintes de mobilité : emettre plus tôt si la distance augmente, plus tard si elle diminue.

Le système doit être assez réactif pour que l'utilisateur ne s'aperçoive de rien ( 100ms).

## Deuxième partie II

# Fonctionnement monotâches

- 1 Scrutation des évènements
- 2 Gestion d'interruptions synchrones
  - Utilisation des interruptions
- 3 Gestion d'interruptions asynchrones
- 4 Protection des structures de données
  - Buffer circulaire
  - Queue
  - Désactivation des interruptions
  - Spin Lock
- 5 Les limites du monotache

# Quelques définitions

- **Temps de réponse** : temps entre un évènement et la fin du traitement de l'évènement.
- Il est possible de formaliser cette définition comme nous le verrons plus tard

# Scrutation des évènements

- Aussi appelé *polling*
- Boucle infinie
- On teste des valeurs des entrées à chaque tour de boucle

```
1 #define sensor1 *((char *) 0x1234)
  #define sensor2 *((char *) 0xABCD)

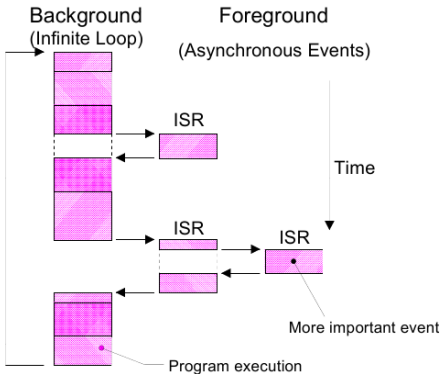
int main() {
    while (1) {
        if (sensor1)
            action1();
        if (sensor2)
            action2();
    }
11 }
```

# Scrutation

- Temps de réponse au évènements en pire cas facile à calculer :  
Pire temps pour parcourir la boucle
- Simple à programmer lorsqu'il y a peu de périphériques (ayant des temps de réaction similaires). On peut les scruter en même temps
- Utilisation du CPU sous optimal. Beaucoup de temps est utilisé pour lire la valeur des entrée. Ceci est particulièrement vrai si les évènements sont peu fréquents
- Si certains évènements entraînent des temps de traitement long ou si il y a beaucoup d'entrées à scruter, le temps de réponse en pire cas peut rapidement devenir très grand
- Tous les évènements sont traité avec la même priorité
- Mauvaise modularité du code. Ajouter des des périphériques revient à repenser tout le système

# Interruptions synchrones

- Appelé aussi Background/Foreground
- Gestion des évènements dans les interruptions



# Interruptions synchrones

Concrètement :

```
#define PTR_DATA ((char *) 0x1234)

void isr() {
    action1(*PTR_DATA);
    *PTR_DEVICE_ACK = 1;
}

int main() {
    9 enable_interrupt(isr, 0x1);
    while(1) {
        ; // Optionnal background computing
    }
}
```

# Interruptions synchrones

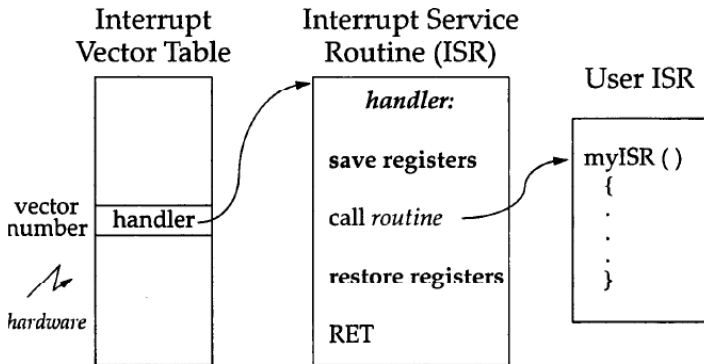
- Temps de réponse au évènements plutôt bon
  - Temps de réponse assez simple à calculer. Somme de
    - Temps de traitement de l'évènement
    - Temps de traitement des évènements de priorité supérieures
    - Temps du changement de contexte (plus ou moins constant)
    - Pire interval de temps ou les interruptions sont désactivée
- Dans un système simple, ca peut se calculer à la louche
- Le temps de réponse en pire cas des calcul en tâche de fond est quasiment identique au traitement par scrutation (attention tout de même à la fréquence maximum des interruptions)

# Qu'est-ce qu'une interruption ?

Il existe trois type d'interruptions :

- Les interruption matérielles :
  - **IRQ** (aussi appelé Interruption externe). Asynchrone. Exemples : clavier, horloge, bus, DMA, second processeur, etc...
  - **Exception**. Asynchrone ou Synchrones. Exemples : Division par zéro, Erreur arithmétique, Erreur d'instruction, Erreur d'alignement, Erreur de page, Breakpoint matériel, Double faute, etc...
- **Logicielle**. Déclenché par une instruction. Synchrones.

# Fonctionnement d'une interruption



Copyright © Wind River Systems, Inc.

# Fonctionnement d'une interruption

Quand une interruption est levée :

- le CPU sauve en partie ou en totalité le contexte d'exécution (principalement le pointeur d'instruction) sur la pile
- Le CPU passe en mode superviseur (nous y reviendrons)
- Le CPU recherche dans l'IVT (*Interrupt Vector Table* aussi appelé IDT, *Interrupt Description Table*) l'ISR (*Interrupt Service Routine*) associée
- Le CPU place le pointeur d'instruction sur l'ISR
- L'ISR traite l'évènement (fin de traitement d'une E/S, etc...)
- L'ISR acquite la réception de l'interruption indiquant qu'une nouvelle donnée peut-être traitée.
- L'ISR restore le (un) contexte

# Fonctionnement d'un PIC

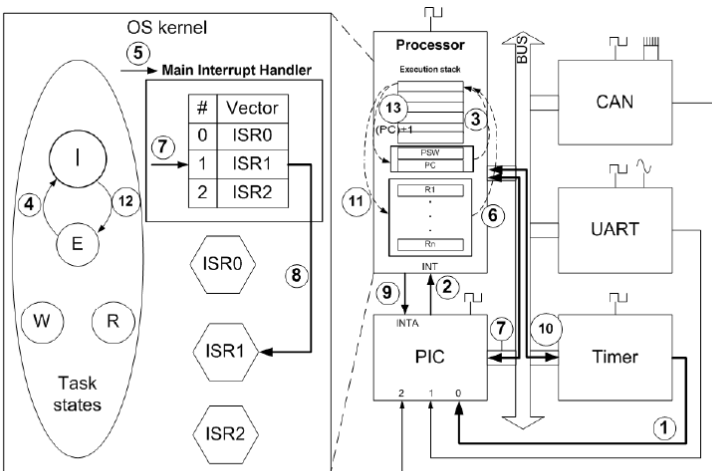
Le PIC (Programmable Interrupt Controller) est un composant matériel permettant la gestion des IRQ. Il peut-être intégré au CPU ou externe (ou à cheval entre les deux...). Il permet en particulier :

- Activer ou de désactiver des IRQ
- De masquer temporairement une IRQ
- De mettre en queue une interruption temporairement masquée
- De contrôler la priorité des interruptions

Il arrive fréquemment qu'un PIC soit multiplexé sur une seule ligne d'IRQ. Dans ce cas, le premier étage d'ISR lit un registre du PIC pour connaître le numéro de l'IRQ. (Cas notoire du 8259A sur les architectures x86)

# Exemple

Exemple classique d'intégration d'un PIC multiplexé sur une IRQ :



E Executing  
R Ready  
W Waiting  
I ISR Running

# Exemple

- 1 Le périphérique *Timer* lève sa ligne d'IRQ
- 2 Le PIC reçoit l'interruption et lève une IRQ du processeur
- 3 Le processeur complète l'instruction courante et sauve le registre d'instruction (PC) et le registre d'état (PSW)
- 4 La tâche courante devient interrompue (Nous y reviendrons)
- 5 Le premier étage d'ISR est appelé
- 6 Le gestionnaire d'interruption complète la sauvegarde des registres
- 7 Le gestionnaire d'interruption demande au PIC quelle interruption à été appelée et il lit dans l'IVT quelle ISR est associée

# Exemple

- 9 Le gestionnaire d'interruption se branche sur l'ISR associée (ici, ISR1)
- 10 L'IRQ du processeur est acquitée. Les autre IRQ peuvent ainsi être levées
- 11 L'ISR1 lit la valeur provenant du *Timer* et acquite l'interruption du *Timer*. Ce périphérique peut de nouveau lever des IRQ.
- 12 Les registres généraux sont restaurés
- 13 Le contexte d'exécution est restauré
- 14 Le registre PC est restauré

# Exemple

Exemple de différence d'approche entre la gestion par scrutation et la gestion par interruption :

Prenons l'acquisition de donnée à partir d'un convertisseur analogique/numérique asynchrone

- Dans le cas du traitement par scrutation, nous allons périodiquement voir si un résultat est arrivé. Beaucoup de temps est consommé pour rien et lorsque le résultat arrive, le traitement du résultats sera retardé
- Une interruption est levée quand une nouvelle donnée est disponible. Le processeur peut alors la traiter.

# Latence des interruptions

- Un périphérique ne génère pas d'IRQ si la précédente n'est pas acquitée (en principe)
- Vu le fait que les interruptions sont souvent multiplexées, les interruptions sont souvent désactivées lors de la première phase de traitement
- Pour des raisons techniques, il est parfois nécessaire de désactiver les interruptions
- Le partage de l'information entre les interruptions et le reste du programme nécessite parfois de désactiver les interruptions (Nous y reviendrons)

Les conséquences :

- Augmente les temps de réponses
- Temps réponse plus difficile à calculer
- Risque de perdre des interruptions (Dans ce cas, une interruption *overrun* est (devrait être) déclenchée)

# Precautions avec les interruption

- Acquiter l'interruption le plus tot possible
  - Rester le moins de temps possible dans dans une interruption
  - Accéder à un minimum de données pour eviter d'avoir à partager des données avec le background
  - Transferer un maximum de traitement hors de l'interruption
- Gestion des interruption asynchrone

# Interruptions asynchrones

- Interruption séparée en deux parties : *top half* et *bottom half*
- On délègue le maximum de traitement au *bottom half*
- Permet de décharger les interruptions
- Permet de plus facilement prendre en compte des interactions entre les évènements (exemple, possibilité d'attendre deux évènements avant d'effectuer une action)

# Interruptions asynchrones

```
#define PTR_DATA ((char *) 0x1234)
```

```
int gotit = 0;
```

```
void isr() {  
    gotit++;  
    *PTR_DEVICE_ACK = 1;
```

7

```
}
```

```
int main() {  
    enable_interrupt(isr, 0x1);
```

```
    while(1) {  
        if (gotit) {  
            gotit--;  
            action1();  
        }  
    }
```

```
    // Optional background computing
```

17

```
}
```

```
}
```

# Exemple de partage de données

Imaginons le code suivant :

```
#define PTR_DATA ((char *) 0x1234)
2 int a = 0;
char t[255];
void isr() {
    t[a++] = *PTR_DATA;
    *PTR_DEVICE_ACK = 1;
}
void main() {
    enable_interrupt(isr, 0x1);
    while(1) {
        if (a)
12     action1(t[--a]);
        // Optionnal background computing
    }
}
```

# Exemple de partage de données

Prenons le cas où  $\mathcal{f}$  traite l'interruption précédente (donc  $a = 1$ ) et qu'une nouvelle interruption est déclenchée :

```
--a; // a = 0;
```

5

```
action1(t[a]);  
// Lecture de t[1] au  
// lieu de t[0]!
```

```
2 t[a] = *PTR_DATA;  
// t[0] est écrasé!  
a++;  
// a = 1 maintenant  
*PTR_DEVICE_ACK = 1;
```

Au lieu de lire correctement la première valeur retournée par l'ISR puis la seconde, nous lisons tout d'abord une valeur aléatoire puis la valeur retournée par la seconde interruption.

# Comment éviter le problème ?

Les problèmes d'accès concurrents se traduisent très souvent par des *race conditions*. C'est à dire des problèmes aléatoires produit par une séquence particulière d'évènements

- Les *race conditions* sont souvent difficiles à reproduire et à identifier
- Les *race conditions* peuvent être latente dans le code et se déclarer suite à une modification de l'environnement externe
- Une race condition coûte chère (difficulté de correction, peut potentiellement atterrir en production)

Comment s'en protéger ?

- Ne pas partager de données avec les interruptions
- Utiliser des accès atomiques
- Utiliser des structures adaptées : buffers circulaires et queues
- Désactiver les interruption lors d'accès à des données partagées

# Buffer circulaire

```
char buf[SIZE];
2 void init() {
    w = r = 0;
}
void write(char c) {
    if ((w + 1) % SIZE == r )
        ; //buffer is full
    buf[w] = c;
    w = (w + 1) % SIZE;
}
12 void read() {
    if (w == r)
        ; //buffer is empty
    ret = buf[r];
    r = (r + 1) % SIZE;
}
```

# Queue

Même fonctionnement que le buffer circulaire, mais avec un tableau de structure.

Il est aussi possible de faire des *queue* d'objets de tailles différentes. Dans ce cas, faire très attention à l'allocation des objets. L'allocation dynamique est rarement une opération très bornée dans le temps et doit être utilisée avec précaution à l'intérieur des interruptions et des tâches temps réelles.

# Désactivation des interruptions

Si l'utilisation de Buffer circulaires ne résoud pas le problème, il est possible de désactiver les interruptions.

La désactivation des interruption peut entraîner des latence dans la gestion des interruption et des perte d'interruptions le cas échéant.

# Cas des interruption en milieu multi coeur

- On ne désactive que les interruptions locales (sur le CPU courant)
- Une interruption peut se produire sur un autre coeur
- Nécessité d'utiliser un mécanisme supplémentaire d'exclusion
- *Spin lock* suvent utilisé pour ce cas.
- Pas beaucoup d'autres choix. Par conséquent les sections critiques dans les interruptions doivent être très limitées

# Spin Lock

Attente active.

Nécessite une instruction assembleur permettant un accès en lecture et une écriture en une instruction :

`test_and_set` affecte le registre d'état en fonction de la valeur du registre et affecte la valeur 1 au registre.

```
4 void lock(int m) {  
    while(atomic_test_and_set(m))  
        ;  
}  
  
void unlock(int m) {  
    m = 0;  
}
```

# Problèmes de la gestion des interruption asynchrone

Nous n'avons pas résolu notre problème récurrent :

- Le partage de l'information entre les interruptions et la boucle principale entraîne des latences

On retrouve certains problèmes que l'on avait avec la scrutation :

- Ne permet pas de prioriser les traitement dans la boucle principale
- Interaction entre les évènements complexe