

Systemes temps-réel embarqués

Brique ROSE

Samuel Tardieu
sam@rfc1149.net

École Nationale Supérieure des Télécommunications

Certains problèmes sont très liés au temps :

- un distributeur de billets ne doit pas mettre 5 minutes à délivrer les billets
- une balance ne doit pas peser en 30 secondes
- un radar ne doit pas mettre 2 secondes à réagir
- un système de freinage ABS ne doit pas mettre plus de 150ms pour acquérir l'information et 1s pour réagir

Caractéristiques :

- exactitude logique (comme tout système) : les sorties sont déterminées en fonction des entrées et de l'état interne
- exactitude temporelle : les sorties sont positionnées au bon moment

- **Temps-réel mou** : un retard dans l'obtention du résultat n'est pas dramatique (distributeur de billets)
- **Temps-réel dur** : un retard dans l'obtention du résultat le rend inutile (détection de missile)
- **Temps-réel ferme** : un retard, s'il arrive très peu souvent, peut être toléré (téléphonie)

La plupart des systèmes temps-réel sont hybrides.

- **Précision** : effectuer certaines opérations à un moment précis (horloge dont l'aiguille avance toutes les secondes)
- **Temps de réponse** : effectuer certaines opérations en un temps maximum (système de freinage ABS) ou avec un temps moyen fixé (distributeur de billets)
- **Rendement** : nombre de requêtes traitées par unité de temps (robot de production dans une usine)

- Un système temps-réel à une seule tâche est simple à définir. Il suffit de répéter indéfiniment la suite de tâches :
 - Attendre un stimulus
 - Agir en fonction du stimulus
- Le dimensionnement du processeur dépend du temps de réponse souhaité.
- Exemples :
 - Un distributeur automatique
 - Une carte à puce

Plusieurs problèmes se posent lorsque plusieurs tâches s'exécutent simultanément :

- accès au processeur
- accès concurrent à la mémoire
- accès aux périphériques

Il faut prévoir un ordonnancement permettant au système de remplir son rôle.

- Les priorités permettent d'organiser les tâches
- Elles peuvent être statiques ou dynamiques
- Dans un système temps-réel, en général
 - une tâche n'est jamais bloquée par une tâche de moindre priorité (inversion de priorité)
 - une tâche ne cède la main à une tâche de même priorité que volontairement (prix du changement de contexte)
- Le système Unix n'est pas temps-réel à la base
- Microsoft Windows n'est pas temps-réel

On peut prévoir le comportement d'un système si on connaît certaines caractéristiques des tâches :

- loi d'arrivée
- temps de traitement

Methodes :

- algorithmes statiques : table d'exécution ou analyse « rate monotonic »
- algorithmes dynamiques (HPF, EDF, LLF, *best effort*)

Avantages :

- l'ordre d'exécution des tâches est connu
- l'accès aux ressources est centralisé

Inconvénients :

- les événements entrants ne peuvent pas être traités rapidement
- les caractéristiques du système doivent toutes être connues à l'avance

Méthode :

- traitement de tâches périodiques
- des priorités statiques sont affectées aux différentes tâches (utilisation de HPF, *Highest Priority First*)
- ces priorités doivent garantir la possibilité d'ordonnancer le système

Inconvénients :

- le rythme d'arrivée des tâches et leur temps d'exécution doivent être bien définis

Rate monotonic Analysis (RMA)

- les tâches ayant les plus courtes périodes doivent être les plus prioritaires
- si un ensemble de tâches ne peut pas être ordonnancé en utilisant l'algorithme RMA, aucun algorithme basé sur des priorités statiques ne permettra de satisfaire les contraintes

Soient :

- C_i le temps d'exécution de la tâche i
- T_i la période de la tâche i
- D_i l'échéance ($D_i = T_i$)
- $U_i = C_i/T_i$ le taux d'utilisation du processeur pour la tâche i
- $U = \sum_{i=1}^n U_i$ le taux d'utilisation du processeur

Si $U \leq n \left(2^{\frac{1}{n}} - 1 \right)$ alors le système est ordonnançable. On peut noter que

$$\lim_{n \rightarrow +\infty} \left(n \left(2^{\frac{1}{n}} - 1 \right) \right) = \ln(2) \approx 0,6931$$

Théorème de la zone critique

Si l'ensemble des tâches respectent leur première échéance, alors toutes les tâches respecteront leurs échéances futures.

On dispose d'un système à deux tâches

- Si $T_1 = 50\text{ms}$, $C_1 = 25\text{ms}$, $T_2 = 100\text{ms}$, $C_2 = 40\text{ms}$:
 - $U_1 = 50\%$
 - $U_2 = 40\%$
 - $U = U_1 + U_2 = 90\%$
 - Ordonnançable si la tâche 1 est plus prioritaire que la tâche 2
- Si $T_1 = 50\text{ms}$, $C_1 = 25\text{ms}$, $T_2 = 75\text{ms}$, $C_2 = 30\text{ms}$:
 - $U_1 = 50\%$
 - $U_2 = 40\%$
 - $U = U_1 + U_2 = 90\%$
 - Système non ordonnançable – la limite théorique d'utilisation du processeur est de $2(2^{\frac{1}{2}} - 1)$ soit moins de 83%

Si, dans un système de tâches périodiques, chaque période est un multiple entier des périodes de durée inférieure, il est possible d'utiliser le processeur à 100%.

- *Earliest Deadline First*
- le travail dont le résultat est nécessaire le plus rapidement est exécuté d'abord

Inconvénients :

- une échéance manquée provoque une avalanche de retards d'échéances

- *Least Laxity First*
- le travail à qui il reste le moins de marge s'exécute d'abord

Inconvénients :

- il faut estimer le temps nécessaire pour chaque travail

- Un sémaphore S est composé de :
 - Un compteur S_n , initialisé lors de la création du sémaphore
 - Une liste de tâches en attente sur ce sémaphore S_t , initialement vide
- Les opérations sont :
 - $P(S)$: prise du sémaphore (demande d'une ressource)
 - $V(S)$: relâchement du sémaphore (libération d'une ressource)

Opérations sur un sémaphore

Lorsqu'une tâche T opère sur un sémaphore,

- $P(S)$
 - Dans tous les cas, décrémenter S_n
 - Si $S_n < 0$, ajouter T en queue de S_t et bloquer la tâche T
 - $P(S)$ est une opération potentiellement bloquante, c'est un cas de **contention** (plus de candidats que de ressources)
- $V(S)$
 - Dans tous les cas, incrémenter S_n
 - Si S_t n'est pas vide, débloquent la première tâche de la liste et la retirer de la liste
 - $V(S)$ n'est jamais une opération bloquante

- S_n a un double rôle
 - lorsque $S_n \geq 0$, il représente le nombre de ressources libres
 - lorsque $S_n < 0$, il représente le nombre d'entrées dans la file d'attente ($S_n = |S_t|$)
- Un verrou est un sémaphore initialisé avec $S_n = 1$
- Le sémaphore doit protéger ses propres structures contre l'accès concurrent : il utilise une instruction **test-and-set** fournie par le processeur et gérée par le matériel

- Une ressource peut être protégée par une ou plusieurs sections critiques utilisant un même verrou.
Dans la section critique, une tâche :
 - peut accéder librement à la ressource
 - peut modifier librement la ressource
 - ne doit pas attendre une autre ressource
- La longueur des sections critiques affecte le temps de réponse du système.

Les sections critiques ne suffisent pas :

- une ressource peut être bloquée pendant longtemps (réseau)
- une tâche de haute priorité peut avoir besoin d'une ressource bloquée

Il y a des risques de blocage : une tâche de haute priorité peut être bloquée par une tâche de plus basse priorité.

Pour éviter les inversions de priorité, on peut utiliser l'héritage de priorité :

- une tâche T_1 a pris une ressource R
- une tâche T_2 plus prioritaire souhaite prendre la ressource R
- on augmente temporairement la priorité de la tâche T_1 à la priorité de la tâche T_2 pour qu'elle libère plus vite la ressource R

Il est possible d'éviter les inversions de priorité en affectant un plafond aux ressources. Une tâche ne peut pas prendre une ressource dont le plafond (calculé *a priori*) est plus important que la priorité de la tâche.

Si une tâche veut prendre une ressource R , elle doit attendre que le plafond de toutes les ressources bloquées soient moins important que sa propre priorité (ou égal).

- Les tâches communiquent
 - par rendez-vous (Ada)
 - par boîte à lettres (Erlang, Marvin, Esterel)
 - par des mécanismes bancales (C/C++ et threads posix, Java)
- Le rendez-vous est construit grâce à une variable conditionnelle, un compteur et un verrou
- La boîte à lettres peut être protégée par un verrou

Variable conditionnelle

- Prenons une variable conditionnelle C et un verrou S
- Opérations :
 - $wait(C, S)$ (S doit être pris auparavant) bloque la tâche courante sur la variable conditionnelle C tout en relâchant S de manière atomique. Quand la tâche sera débloquée, elle aura de nouveau acquis S .
 - $signal(C)$ débloque une tâche en attente sur la variable conditionnelle C
 - $broadcast(C)$ débloque toutes les tâches en attente sur la variable conditionnelle C (une par une, à cause du sémaphore S qui doit être réacquis)

Le rendez-vous

- On souhaite amener deux tâches (au moins) à un endroit donné
- On souhaite les laisser bloquées jusqu'à ce que toutes soient arrivées au rendez-vous
- On construit un compteur avec le nombre de tâches, protégé par un verrou. Chaque tâche qui arrive :
 - prend le sémaphore
 - décrémente le compteur
 - broadcast la variable conditionnelle si le compteur est nul et relâche le sémaphore, ou se met en attente sur la variable conditionnelle sinon
 - avant de relâcher le sémaphore, tout le monde réincrémente le compteur

Rendez-vous en threads Posix

```
int n = 2;      /* Nombre de t^e2ches */
pthread_cond_t *cond;
pthread_mutex_t *verrou;

void rendez_vous ()
{
    pthread_mutex_lock (verrou);
    n = n - 1;
    if (n == 0) pthread_cond_broadcast (cond);
    else pthread_cond_wait (cond, verrou);
    n = n + 1;
    pthread_mutex_unlock (verrou);
}
```

- Tâche A

```
accept RendezVous (I : in out Integer) do  
  I := I + 1;  
end RendezVous;
```

- Tâche B

```
A.RendezVous (V);
```

- C'est beaucoup plus simple lorsque le langage prévoit la synchronisation directement.

- Le noyau Linux n'est pas temps-réel
- Les extensions temps-réel
 - utilisent les services du noyau pour les portions non critiques (chargement d'un module sur le disque)
 - n'utilisent pas les services du noyau pour les portions critiques
 - considèrent le noyau comme une tâche de plus basse priorité
 - utilisent des files de messages entre les différentes tâches