Institut
Mines-Télécom

Systèmes d’exploitation embarqués

ELECINF344

Samuel TARDIEU <sam@rfc1149.net>
Mars 2015

Plan

Introduction

Gestion de la mémoire

Gestion de la concurrence

Quelques OS embarqués

Système d'exploitation

Un système d'exploitation (*Operating System*, ou *OS*) fait le lien entre le logiciel (application) et le matériel :

- abstrait certaines caractéristiques du matériel ;
- fournit des services communs (accès aux ressources, synchronisation, gestion de fichier) ;
- offre des possibilités de tests et de traces.

Système embarqué

Généralement, un système embarqué :

- doit être le moins cher possible ;
- dispose de ressources limitées ;
- ne doit pas consommer d'énergie inutilement (batterie).

TELECOM ParisTech

Un OS est-il obligatoire ?

Absolument pas !

- Certains langages sont un OS à eux tout seul (Forth).
- Certains langages incluent des options de concurrence plus ou moins avancées (Ada, Java).
- Certains projets sont très simples.

Un OS est-il utile ?

Absolument !

- La plupart des programmes embarqués ont des contraintes comparables (concurrence, temps-réel, USB, TCP/IP, fichiers).
- Il est plus facile d'utiliser une API portable plutôt que de recoder des fonctionalités de base.
- Pourquoi réinventer la roue systématiquement ?
- Pourquoi s'interdire de changer de CPU par la suite ?

Un OS résout-il tous les problèmes ?

Non !

- La gestion de la mémoire reste à la charge du développeur (d'où un conseil : ne gérez pas la mémoire et privilégiez les allocations statiques).
- Les tâches ne s'ordonnent pas toutes seules : il faut en choisir les priorités statiques et dynamiques.
- Un OS n'empêche pas les bugs dans le code, mais bénéficie généralement de plus d'utilisateurs que votre code.

TELECOM ParisTech

Plan

Introduction

Gestion de la mémoire

Gestion de la concurrence

Quelques OS embarqués

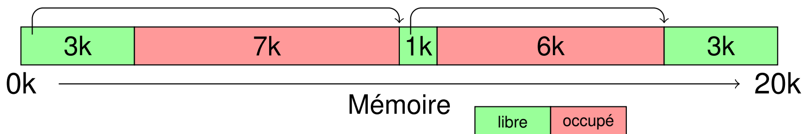
TELECOM ParisTech

Gestion de la mémoire

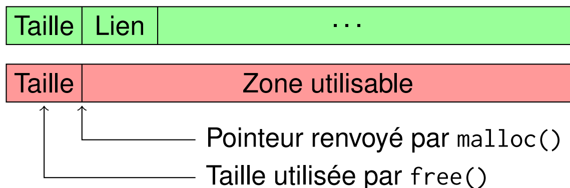
- La plupart des microcontrôleurs embarquent quelques k de mémoire :
  - 256k sur les STM32F427IG ;
  - 16k sur les nRF51822-QFAA-R7 ;
  - 25 octets sur le PIC12F508.
- Ajouter de la RAM externe est coûteux :
  - utilisation d'entrées-sorties supplémentaires sur le processeur ;
  - bus complet pas forcément disponible ;
  - complication du routage ;
  - intégrité du signal.
- Il faut gérer précautionneusement cette ressource précieuse.

Gestion dynamique de la mémoire

- Utilisation d'une liste chaînée des blocs libres (*free-list*)

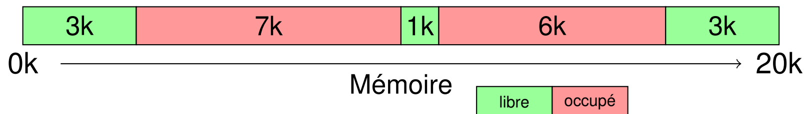


- Stockage de la taille réservée en mémoire (la taille libérée n'est pas passée à `free()`.



Fragmentation

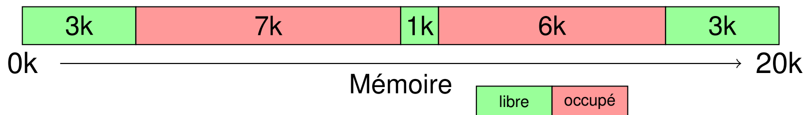
Au cours de son utilisation, la mémoire disponible peut devenir fragmentée.



Comment allouer 5k alors que seuls deux blocs non contigus de 3k et un de 1k sont disponibles ?

Politiques d’allocation

Dans la situation suivante, dans quel bloc allouer une zone de 600 octets demandée par le programme ?



Plusieurs stratégies possibles :

- *Best fit*
- *Worst fit*
- *First fit*
- *First fit* équivalent à une des deux premières solutions en triant la liste des blocs libres

Gestion de la libération

Plusieurs stratégies possibles :

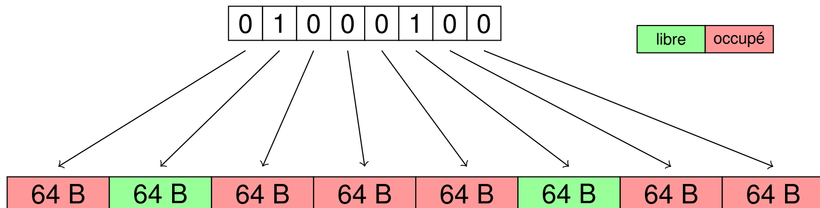
- Agrégation des blocs libres, peut nécessiter un tri de la liste ; peu déterministe.
- Libération sans agrégation des blocs libres, peut nécessiter un tri de la liste.
- Pas de libération.

Toutes ces stratégies sont couramment utilisées. La dernière permet l'allocation dynamique en début de programme, qui ne commencera ses véritables fonctions qu'après que l'ensemble des allocations aient été effectuées.

Gestion par *bitmaps*

La mémoire peut-être gérée avec des *bitmaps* :

- blocs de taille fixe et contigus ;
- un bit par bloc indique si le bloc est libre ou non ;
- possibilité d'utiliser plusieurs zones avec des blocs de taille différente.



Recherche dans un *bitmap*

Comment, dans un mot machine, isoler facilement le bit de poids le plus faible parmi les bits à 1 ?

0	1	0	0	0	1	0	0	`word`
0	0	0	0	0	1	0	0	`lowest_bit(word)`

Recherche dans un *bitmap*

Comment, dans un mot machine, isoler facilement le bit de poids le plus faible parmi les bits à 1 ?

0	1	0	0	0	1	0	0	word
0	0	0	0	0	1	0	0	lowest_bit(word)

```
inline unsigned int lowest_bit(unsigned int word)
{
  return ((word ^ (word - 1)) >> 1) + 1;
}
```

Recherche dans un *bitmap*

Comment, dans un mot machine, isoler facilement le bit de poids le plus faible parmi les bits à 1 ?

0	1	0	0	0	1	0	0	word
0	0	0	0	0	1	0	0	lowest_bit(word)

```
inline unsigned int lowest_bit(unsigned int word)
{
  return ((word ^ (word - 1)) >> 1) + 1;
}
```

0	1	0	0	0	1	0	0	word
0	1	0	0	0	0	1	1	word - 1
0	0	0	0	0	1	1	1	^
0	0	0	0	0	0	1	1	>> 1
0	0	0	0	0	1	0	0	+ 1

Allocation statique

L'absence d'allocation dynamique a des (énormes) avantages :

- détermination de la position définitive de chaque bloc lors de l'édition de liens et temps d'accès réduit ;
- vérification de la disponibilité de la quantité nécessaire de mémoire lors de l'édition de liens ;
- aucune possibilité de fragmentation ou de manque de mémoire lors de l'exécution.

Cette solution doit être privilégiée lorsque c'est possible. Cela peut influencer le choix du système d'exploitation.

Utilisation de la MMU

L'utilisation d'une MMU (*Memory Management Unit*) permet :

- la protection des zones mémoire ;
- la réduction de la fragmentation par l'utilisation de pages et de la correspondance entre adresse logique et adresse physique ;
- la possibilité de disposer de zones *thread-local* sans indirection supplémentaire.

Tous les systèmes n'utilisent pas la MMU même lorsqu'elle est présente, pour des raisons de simplicité ou de performances.

TELECOM ParisTech

Plan

Introduction

Gestion de la mémoire

Gestion de la concurrence

Quelques OS embarqués

Architectures et parallélisme

On trouve plusieurs types de systèmes embarqués :

- mono-cœur ;
- SMP (*symetric multiprocessing*) ou multi-cœur ;
- NUMA (*non-uniform memory architecture*).

Même dans les architectures mono-cœur, on souhaite souvent exécuter plusieurs activités de manière concurrente.

Boucle principale

Le schéma le plus simple est celui de la boucle principale :

```
static void tache_1()
{
  if (capteur_1_actif())
    reagir_a_capteur_1();
}

...

int main()
{
  for (;;) {
    tache_1();
    ...
    tache_n();
  }
}
```

Inconvénients :

- gaspillage des ressources par l'utilisation systématique du *polling* ;
- pas de gestion de priorité ou de fréquence relative.

Boucle événementielle

On peut également attendre un événement signalé par le matériel :

```
int main()
{
  for (;;) {
    switch (wait_for_event()) {
    case capteur_1:
      reagir_a_capteur_1();
      break;
    ...
    case capteur_n:
      reagir_a_capteur_n();
      break;
    }
  }
}
```

Inconvénients :

- pas de priorisation des événements ;
- pas de possibilité de faire de longs calculs sans bloquer la gestion des autres événements.

Fonctionnement sur interruptions

On peut avoir :

- une tâche principale qui s'exécute en permanence ;
- des routines lancées lorsqu'une interruption survient.

C'est ainsi que fonctionnent des systèmes mono-tâche pour « émuler » le multi-tâches (sur interruption de timer par exemple).

Inconvénients :

- une seule tâche principale.

Changement de contexte

On souhaiterait pouvoir :

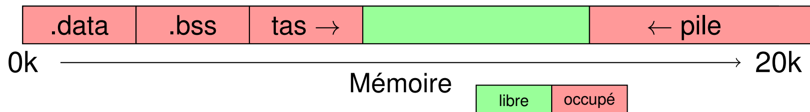
- sauvegarder l'état d'un traitement donné pour en effectuer un autre et retrouver son état ;
- effectuer des traitements longs en donnant leur chance aux autres tâches à réaliser.

Pour cela, chaque tâche doit disposer de son contexte :

- l'état des registres du processeur dont le pointeur ordinal (*program counter*) et le pointeur de pile (*stack pointer*) ;
- la pile d'appel des sous-programmes, qui doit donc être propre à chaque tâche.

Allocation de la pile (mono-tâche)

Dans une architecture mono-tâche, la pile et le tas peuvent croître en sens inverse.



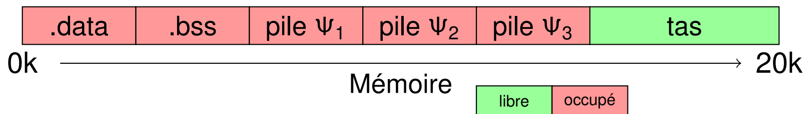
Tant que les deux zones ne se rencontrent pas, tout va bien.

TELECOM ParisTech

Allocation de la pile (multi-tâches)

Dans une architecture multi-tâches, chaque tache Ψ_i nécessite une pile d'exécution. L'allocation de cette pile est critique :

- trop petite, elle causera une corruption mémoire ;
- trop grande, elle consommera trop de mémoire.



La taille à choisir dépend des profondeurs d'appel de chaque tâche et de la taille occupée par les variables locales.

Synchronisation

L’utilisation d’un système multi-tâches peut poser des problèmes de synchronisation :

```c
void transfert(compte *origine, compte *destination, unsigned int montant)
{
  if (origine->solde >= montant) {
    /* (1) */
    origine->solde      -= montant;
    /* (2) */
    destination->solde += montant;
  }
}
```

Si cette routine est appelée deux fois avec les mêmes paramètres simultanément et que le changement de contexte se fait en (1), la vérification peut s’avérer incorrecte. Si une autre tâche effectue un total des soldes alors que le changement de contexte s’est fait en (2), le total sera inférieur à ce qu’il devrait être.

Coroutines

Les coroutines sont un modèle de parallélisme coopératif :

- chaque tâche indique (avec `yield()`) qu'elle accepte de donner la main à une autre ;
- les problèmes de synchronisation n'existent pas, les points de synchronisation étant placés par l'utilisateur.

Certains modèles de coroutines sans pile existent :

- la consommation mémoire est réduite ;
- les variables locales sont interdites ;
- `yield()` ne peut être appelé que depuis le sous-programme principal de la coroutine ;
- sur certaines architectures avec une pile dédiée (PIC18F), le changement de contexte est bien plus rapide.

Priorités

Les priorités permettent de choisir la prochaine tâche à exécuter :

- les priorités peuvent être statiques ou dynamiques ;
- le temps maximum entre deux changements de contexte indique le temps de réaction à une condition extérieure ;
- on peut créer une *idle task* de priorité minimale qui met le processeur en veille en attendant qu'un événement modifie le système ;
- cette même *idle task* peut également effectuer des opérations de nettoyage (agrégation des blocs mémoire libres) ;
- le passage régulier dans l'*idle task* peut indiquer une non-surcharge du système et servir à signaler un *watchdog*.

Systèmes préemptifs

Un système multi-tâches est dit « préemptif » lorsqu'une tâche peut être interrompue sans l'avoir elle-même demandé (yield) ou autorisé (par un appel au système) :

- réaction plus rapide à des événements asynchrones ;
- réveil d'une tâche plus prioritaire à l'expiration d'un délai fixé (alarme) ;
- possibilité ou non de *round-robin* entre des tâches de même priorité (quantum de temps) ;
- nécessité d'utiliser des outils de synchronisation pour utiliser des données communes ou communiquer entre les tâches.

TELECOM ParisTech

Section critique

Une section critique représente un ou plusieurs chemins de code dans lesquels une seule tâche peut se trouver à la fois.

Implémentations possibles :

- inhibition temporaire des interruptions ;
- inhibition temporaire de l'ordonnanceur préemptif, si les routines d'interruption n'utilisent pas les données ;
- utilisation d'un verrou, avec ou sans réentrance, avec ou sans héritage de priorité.

Dans toutes ces solutions, on risque de bloquer une tâche prioritaire.

Sémaphore

Un sémaphore représente un ensemble de ressources et comprend, outre son initialisation, deux opérations :

- `V()`, `release()` ou `signal()` : créé une ressource, jamais bloquant.
- `P()`, `take()` ou `wait()` : demande une ressource, et bloque l'appelant si aucune n'est disponible ; ne revient de l'appel que lorsque la ressource a été acquise.

Une ressource n'appartient à personne. Les opérations de création ou destructions peuvent être effectuées par des threads différents.

Verrou

Un verrou protège une section critique, et est pris puis libérer par un thread. Seul un thread peut posséder le verrou à un moment donné.

En interne, un verrou possède une file d'attente contenant les références des threads en attente du verrou.

Il est possible d'implémenter un verrou à base de sémaphore, mais le verrou est une structure de données plus simple et, lorsqu'il est pris, appartient à un thread, qui peut être identifié en cas de problème.

Problème potentiel

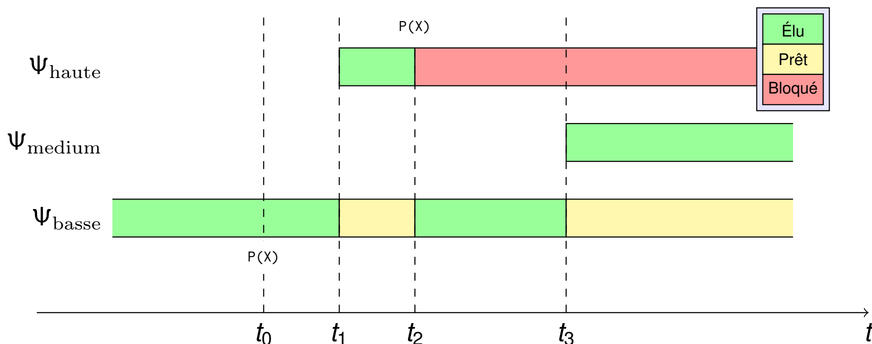
Imaginons la situation suivante :

- un verrou `X` ;
- une tâche Ψ_{basse} de basse priorité acquiert `X` à t_0 ;
- une tâche Ψ_{haute} de haute priorité démarre à t_1 et réclame `X` à t_2 ;
- une tâche Ψ_{medium} démarre à t_3 (avant que la tâche Ψ_{basse} ait relâché `X`) et dure très longtemps.

On peut arriver à une situation où la tâche Ψ_{medium} bloque de par sa seule existence la tâche plus prioritaire Ψ_{haute} sans pour autant posséder de ressource dont cette dernière a besoin pour progresser.

Inversion de priorité

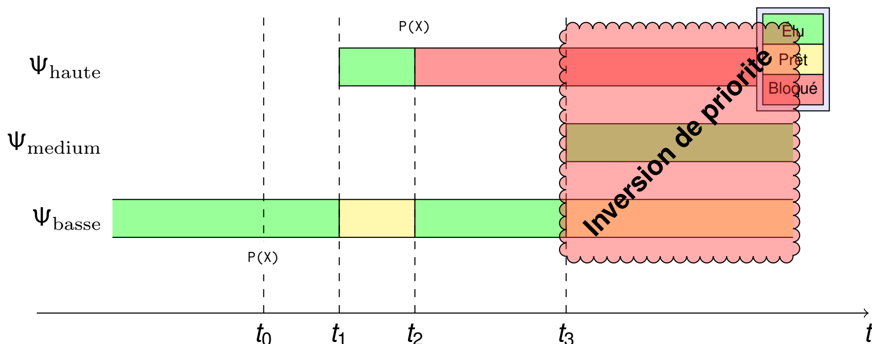
L'inversion de priorité est une situation dans laquelle une tâche moins prioritaire bloque, indirectement, une tâche plus prioritaire, en empêchant la libération d'un verrou.



TELECOM ParisTech

Inversion de priorité

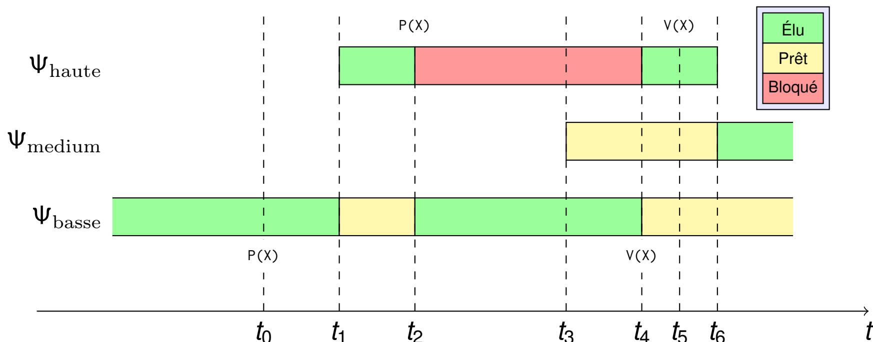
L'inversion de priorité est une situation dans laquelle une tâche moins prioritaire bloque, indirectement, une tâche plus prioritaire, en empêchant la libération d'un verrou.



TELECOM ParisTech

Héritage de priorité

On utilise alors l'héritage de priorité : lorsqu'une tâche plus prioritaire attend un verrou acquis par une tâche moins prioritaire, cette dernière prend la priorité de la première.



Interblocage

Si des tâches utilisent plusieurs sémaphores ou verrous et cherchent à réserver les ressources au même moment, on peut aboutir à des interblocages :

- un *deadlock* lorsqu'aucune des tâches ne peut progresser ;
- un *livelock* lorsque les deux tâches progressent mais passent leur temps uniquement à réserver (et libérer) les ressources.

Certains systèmes détectent les situations de *deadlock* en phase de développement.

Synchronisation et interruptions

Les routines d'interruption (*Interrupt Service Routine* ou ISR), prioritaires, empêchent la progression normale du programme et limitent la gestion par priorités. Pour cela, on divise généralement le traitement en deux parties :

FLIH (*first-level interrupt handler*), consistant à débloquer une tâche qui effectuera le traitement complet de l'interruption et à enregistrer sa prise en compte au niveau matériel ;

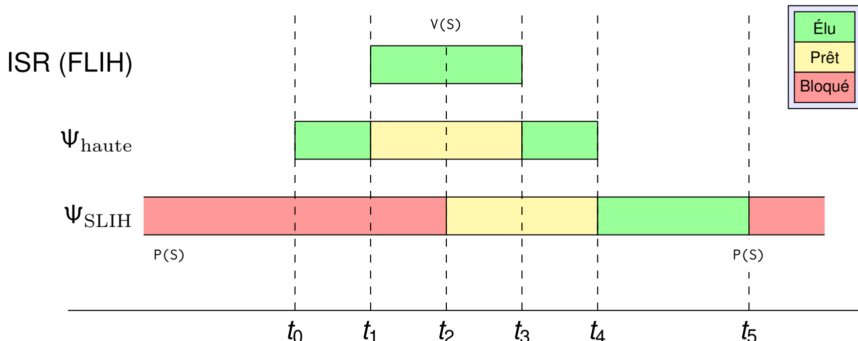
SLIH (*second-level interrupt handler*), tâche ordinaire, disposant de sa priorité propre, qui effectue le traitement, possiblement long, de la condition signalée.

Un événement peu important sera acquitté rapidement au niveau du matériel mais sera possiblement traité beaucoup plus tard lorsqu'il ne restera rien de plus important à faire.

Synchronisation et interruptions

La signalisation est faite à l'aide d'un sémaphore `S` :

- le FLIH donne le sémaphore : `V(S)` (non bloquant) ;
- le SLIH consomme le sémaphore : `P(S)` (potentiellement bloquant).



Différents sémaphores et verrous

Dans les systèmes embarqués, on trouve généralement différents types de sémaphores :

- sémaphores dont le nombre de ressources est plafonné ou non, utilisés principalement pour la synchronisation (FLIH/SLIH ou entre tâches) ;
- verrous avec ou sans héritage de priorité ;
- verrous multi-entrées avec héritage de priorité.

L'opération bloquante `P(S)` sur une entité `S` est généralement assortie d'un *timeout* :

- timeout à zéro : retour immédiat ;
- timeout non nul : temps d'attente limité ;
- timeout « infini » : appel bloquant.

TELECOM ParisTech

Files d’attente

Pour passer des informations de manière protégée, les files d’attente (ou *queue*) permettent de déposer et de récupérer de manière atomique des données ordonnées.

- Chaque file d’attente est créée avec éventuellement une taille maximale et un type de données.
- L’écriture et la lecture sont bloquantes, suivant un modèle producteur/consommateur.
- Les opérations bloquantes sont assorties d’un *timeout*.

Un FLIH peut placer, en mode non-bloquant, certaines données à traiter dans une file d’attente qui sera consultée et vidée par le SLIH.

Ressources et priorités

Lorsqu'une tâche Ψ_1 libère une ressource sur laquelle une tâche Ψ_2 est en attente, Ψ_2 passe immédiatement dans l'état prêt. Si Ψ_2 est plus prioritaire que Ψ_1, cela induit un transfert de contrôle immédiat (changement de contexte) de Ψ_1 vers Ψ_2.
Les raisons d'un changement de contexte sont donc :

- la disponibilité d'une ressource sur laquelle une autre tâche était en attente, depuis une autre tâche ou une routine d'interruption (FLIH) ;
- l'expiration d'un délai, qui consiste en fait à la libération d'une ressource déclenchée depuis une interruption liée à un *timer*, ce qui nous ramène dans le premier cas.

Une tâche en attente d'une ressource ne consommera pas inutilement de temps CPU.

TELECOM ParisTech

Exemple : afficheur LCD

Un microcontrôleur pilote un afficheur LCD en lui envoyant des octets correspondant :

- à un caractère à afficher à la position courante du curseur ;
- à un ordre de déplacement spécifiant la ligne et la colonne.

On souhaite que plusieurs tâches puissent afficher à des endroits différents de l'écran.

Exemple : afficheur LCD

On utilise les entités suivantes :

- une file d'attente contenant les données à envoyer à l'afficheur LCD ;
- une tâche recevant successivement les octets de la file d'attente et les envoyant à l'afficheur ;
- un verrou permettant un accès exclusif à la file d'attente, pour que le remplissage se fasse de manière cohérente.

Ainsi, les différentes chaînes de caractère à afficher ne peuvent pas se mélanger.

Synchronisation et priorités

Si la tâche qui gère l'afficheur est plus prioritaire que la tâche qui souhaite afficher quelque chose, l'affichage peut commencer dès l'entrée du premier octet dans la file d'attente et n'être limité que par les performances de l'afficheur lui-même.

Si les priorités sont égales, il est conseillé de donner la main au consommateur, afin qu'il vide la file d'attente et limite les inversions de priorité par la suite.

TELECOM ParisTech

Système temps-réel

Un système est dit temps-réel lorsque chaque événement est traité dans un délai maximum connu à l'avance :

- Un système réagissant la plupart du temps en 100ns mais dans 0,001% des cas en un temps non borné n'est pas temps-réel, bien qu'il soit rapide.
- Un système réagissant systématiquement en moins de 10s à un événement est temps-réel, bien qu'extrêmement lent.

TELECOM ParisTech

Catégorisation des systèmes temps-réel

Il existe plusieurs types de systèmes temps-réel :

- temps-réel dur : un résultat arrivant après l'échéance est inutile (un *pacemaker* qui ne réagirait pas à temps) ;
- temps-réel mou : un résultat arrivant après l'échéance induit des performances dégradées (omissions d'images dans un décodeur vidéo).

La plupart des systèmes nécessitant du temps-réel comprennent un mélange de trois sous-composants :

- domaine temps-réel dur pour les opérations critiques ;
- domaine temps-réel mou ;
- domaine non-temps-réel, pour l'écriture des fichiers de traces par exemple.

Délais et tâches périodiques

Une tâche peut demander à attendre pendant un certain délai :

- relatif, c'est-à-dire un certain temps ;
- absolu, c'est-à-dire jusqu'à une date donnée.

Une tâche périodique utilisera l'un ou l'autre selon ses besoins :

- un délai absolu permet d'obtenir une exécution à un moment précis indépendamment des retards subis lors d'itération précédentes (gestion d'une horloge) ;
- un délai relatif permet d'espacer des événements d'un intervalle de temps donné (*keep-alive* sur un lien réseau).

Une tâche périodique est caractérisée par sa fréquence et son temps d'exécution d'une itération.

Tâches périodiques et échéances

Étant donné un ensemble fini de tâches périodiques Ψ_i arrivant à intervalles T_i et nécessitant un temps d'exécution C_i, il est parfois possible de garantir qu'aucune tâche n'accumulera de retard en leur affectant des priorités statiques, à l'aide de l'algorithme RMS (*rate monotonic scheduling*).
On peut toujours trouver un tel jeu de priorités si

$$U = \sum_{i=1}^{n} \underbrace{\frac{C_i}{T_i}}_{u_i} \leq n\left(\sqrt[n]{2} - 1\right)$$

On remarquera qu'on a $\lim_{n->+\infty} n(\sqrt[n]{2} - 1) = \ln 2 \approx 69,31\%$. Cela signifie qu'avec un système avec une charge des tâches temps-réel U inférieure à 69,31% on trouvera toujours un moyen d'ordonnancer un tel jeu de tâches à base de priorités statiques.

TELECOM ParisTech

Précautions à prendre

Lors de l'utilisation de l'algorithme RMS, deux précautions particulières doivent être prises :

- Le temps nécessaire aux changements de contexte doit être ajouté au temps d'exécution des tâches.
- Si des moyens de synchronisation sont utilisés, ils doivent implémenter l'héritage de priorité pour éviter les inversions de priorité.

De plus, si toutes les tâches réussissent à remplir leur première échéance (c'est-à-dire à terminer leur premier cycle avant l'arrivée suivante), on peut prouver qu'elles y parviendront systématiquement.

Tâches sporadiques

Une tâche sporadique représente l'exécution d'un code en réaction à un événement. Elle est caractérisée par son temps d'exécution et l'intervalle de temps minimal entre deux exécutions.

On peut faire rentrer les tâches sporadiques dans l'algorithme RMS en les transformant en tâches périodiques. Si nécessaire, il faut adopter une politique spécifique en cas de déclenchement trop fréquent :

- ignorer les déclenchements supplémentaires (en reportant éventuellement une erreur) ;
- sauver dans une file d'attente les déclenchements supplémentaires pour leur faire respecter l'intervalle minimal d'inter-arrivée.

TELECOM ParisTech

Plan

Introduction

Gestion de la mémoire

Gestion de la concurrence

Quelques OS embarqués

TELECOM ParisTech

Linux

- logiciel libre ;
- grand nombre d'architectures de processeurs supportées ;
- disponibilité d'un grand nombre de gestionnaires de périphériques (*device drivers*), et excellente documentation disponible pour en écrire de nouveaux ;
- système multi-tâches préemptif et multi-utilisateur (utile pour la protection du système) ;
- pas de garantie de temps-réel dur par défaut ;
- nécessite beaucoup de RAM.

RTEMS (par OAR)

- logiciel libre, disponible pour de nombreuses architectures ;
- exécutif léger, se combinant lors de la compilation et de l'édition de liens avec l'application de l'utilisateur ;
- système temps-réel multi-tâche préemptif ;
- support de la programmation concurrente en Ada ;
- implémente tous les services POSIX d'un système mono-processus ;
- très utilisé dans le milieu des expérimentations physiques, notamment en milieu spatial ;
- utilisé dans le projet *Mars Reconnaissance Orbiter*.

FreeRTOS

- logiciel libre ;
- supporte un grand nombre de micro-contrôleurs ;
- système temps-réel multi-tâches préemptif, coroutines sans pile ou les deux à la fois ;
- se combine avec l'application finale lors de la compilation et de l'édition de liens ;
- très petit, très rapide et très bien documenté ;
- possède deux versions supplémentaires :

  OpenRTOS : licence ne nécessitant pas d'indiquer qu'on utilise FreeRTOS ni de distribuer les sources de FreeRTOS ;

  SafeRTOS : version certifiée selon plusieurs normes.

ChibiOS/RT

- logiciel libre ;
- supporte un grand nombre de micro-contrôleurs ;
- système temps-réel multi-tâches préemptif ;
- se combine avec l'application finale lors de la compilation et de l'édition de liens ;
- très petit et très rapide ;
- initialisations statique possible de toutes les structures de données ;
- possède un HAL (*hardware abstraction layer*) permettant de s'abstraire des opérations de bas niveau ;
- se charge des opérations d'initialisation du système ;
- s'intègre avec d'autres logiciels libres (gestion de fichiers, du réseau, etc.).

Etc., etc., etc.

Il existe beaucoup d'autres systèmes d'exploitation pour systèmes embarqués non décrits ici, entre autres :

- Android, iOS, Windows Phone 8, BlackBerry 10 (systèmes lourds : téléphones, tablettes, livres électroniques, télévision connectée)
- Contiki (réseaux de capteurs) ;
- eCos (micro-satellites) ;
- VxWorks (transport, avionique, robotique, équipements réseau, imprimantes).

TELECOM ParisTech