

***Le système  
RTEMS***

***Bertrand Dupouy***

***Plan***

- Présentation
- Matériel supportés
- Gestion des tâches
- Gestion de la mémoire
- Communications entre threads
- Timers
- Implantation de pilotes de périphériques
- Réseau
- outils de mise au point
- Divers

## Le système RTEMS

---

### Présentation

<http://www.oarcorp.com>

- Caractéristiques :
  - open-source : on a accès à tous les sources, on peut les modifier, les distribuer
  - c'est un exécutif, pas un système : l'exécutif est « linké » avec l'application et donne un exécutable qui doit être chargé sur la cible,
  - avantage d'un exécutif : plus petit qu'un système (on ne charge que les primitives dont on a besoin), RTEMS peut tourner sur une dizaine de K octets
  - RTEMS est arrivé à maturité
  - Il utilise l'environnement de développement GNU/gcc :
    - Nombreux compilateurs croisés, pratiquement pour toutes les architectures 32 bits
    - Nombreux formats de fichiers exécutables,
    - `remote gdb`
    - Développements sur RTEMS en C, C++, Ada

## Le système RTEMS

---

### Présentation

- Conception modulaire par séparation des :
  - API (RTEMS ou POSIX),
  - Bibliothèques (managers),
  - CPU (Types de processeurs),
  - boards (notion de *board support package* ou BSP)

Application
RTEMS avec les <b>seuls</b> managers nécessaires
Fonctions dépendantes du processeur
Fonctions dépendantes du <i>BSP</i>
Matériel

- Matériels supportés :
  - Famille Motorola MC68xxx et PowerPC, Intel i386 et i960, MIPS, HP PA-RISC, sur toutes sortes de *boards*
  - Ces cibles peuvent être utilisées en multiprocesseur, sans migration

### *Présentation*

- Caractéristiques :
  - RTEMS propose plusieurs API, dont POSIX 1003.1b
  - Les primitives systèmes sont regroupées par type dans des « *managers* »
  - On crée un exécutable en « linkant » le code de l'application avec le noyau, on indique, pour chaque application, quels managers sont utilisés
  - La mise au point peut se faire à distance via une ligne série ou un câble Ethernet
  - Le noyau fonctionne sur des architectures multiprocesseurs, mais les tâches ne peuvent pas migrer
  - Il existe une version Unix qui permet de simuler
- Espace disque pour l'environnement de développement :
  - gcc et bibliothèques : 250 M octets
  - source de RTEMS = environ 95 M octets

### *Gestion des Tâches*

- Pas d'espaces d'adressage séparés, les tâches sont donc des threads
- API spécifique et API POSIX.
- Politique d'ordonnancement :
  - event driven, basée sur les priorités (255 niveaux, 1 est maximum, l'inverse avec POSIX...), préemptive
  - RMS,
  - tâches sporadiques avec l'API POSIX
- Attention : il faut définir le nombre maximal de threads pour chaque application, ce nombre dépend de la mémoire disponible
- Les tâches peuvent être créées et détruites dynamiquement

### ***Gestion Des tâches (suites)***

- Pas de protection des espaces d'adressage,
- Gestion mémoire dynamique par blocs allocation de blocs de taille variable (*memory chunks*), (malloc et free),
- Gestion par le *Region Manager*,
- Communications entre tâches : l'API POSIX 1003.1b IPC est implantée

### ***Synchronisation : Sémaphores***

- Verrous, sémaphores, avec ou sans timeout, nommés ou non, géré par le *Semaphore Manager*
  - locaux, globaux
  - opérations P et V (Dijkstra) : `rtems_semaphore_obtain` et `rtems_semaphore_release`, option NOWAIT
  - opérations Init (Dijkstra) : `rtems_semaphore_create` options : LOCAL/GLOBAL, PRIO : PCP/PIP/STANDARD, FA : FIFO/PRIORITY
  - API POSIX implantée pour les verrous (`pthread_mutex_lock`, ...),
- Gestion des priorités :
  - pour les verrous locaux seulement
  - PIP (Priority Inheritance Protocol), héritage de priorité :
  - PCP (Priority Ceiling Protocol), priorité plafonnée,

### *Synchronisation : Variables conditionnelles*

- servent à mettre un thread en attente de vérification d'une condition,
- implantation de l'API POSIX : association d'un mutex et d'une variable dite conditionnelle
- fonctions de gestion :

```
pthread_cond_init(&VarCond, NULL),  
pthread_cond_destroy(&VarCond),  
pthread_cond_wait(&VarCond, &Verrou),  
pthread_cond_timedwait(&VarCond, &Verrou, &Tempo),  
pthread_cond_signal(&VarCond),  
pthread_cond_broadcast(&VarCond)
```

- le wait est **toujours** bloquant, à la différence d'une opération P sur un sémaphore .Il fait passer le thread à l'état bloqué ET rend le verrou de façon **atomique**.  
Quand le thread sort de l'état bloqué sur un signal ou broadcast, il **essaie de reprendre** le verrou
- l'événement de réveil (`signal`, `broadcast`) n'est pas mémorisé : si aucun thread ne l'attend, il est **perdu** (différent de V sur un sémaphore)

### *Les Messages*

- Gérés par le « Message Manager » :
- Primitives bloquantes on non bloquantes
- API POSIX disponible :

Fonction	Description
mq_close	Fermer une file de messages
mq_getattr	Renvoie les caractéristiques d'une file de messages
mq_open	Ouvrir une file de message
mq_receive	Extraire un message d'une file
mq_send	Déposer un message dans une file
mq_setattr	Changer les attributs d'une file
mq_unlink	Détruire une file de messages

## Le système RTEMS

---

### *Les signaux, timers, Events, interrupts*

- « *Signal Manager* ».
  - gestion par ASR définie par le thread courant qui est exécutée quand le signal est délivré au thread
- API POSIX :

Fonction	Description
Sigqueue	Queue a signal to a process
Sigwaitinfo	Attendre un signal et une info.
Sigtimedwait	Attendre un signal avec une échéance

- La résolution du timer (durée du tic) peut être définie au niveau du BSP
- Les interruptions ne sont pas converties en signaux ou autres événements, elles sont gérées par des fonctions C associées au vecteur d'interruption:
- Les fonctions qui gèrent les IT s'appellent des ISR et sont attachées aux vecteurs d'interruption par la primitive

```
rtems_interrupt_catch()
```

## Le système RTEMS

---

### *Ordonnancement RMS*

- On va créer une période à l'intérieur d'une tâche :
  - appel à `rate_monotonoc_create`,
  - elle est implanté sous forme d'un *Period Control Block*,
  - à chaque PCB est associé un identificateur unique,
  - le PCB contient l'état de la période, initialisé à *inactive*
- Changements d'états pour une période par appel à `rate_monotonoc_create` :
  - ni active, ni expirée, alors initialisée à : *period ticks* et retour à tâche courante,
  - si active, alors la tâche est bloquée pendant le reste de la période courante, à l'expiration de celle-ci, la période est réinitialisée et la tâche redémarre,
  - si la période a expiré avant l'appel à `rate_monotonoc_period`, alors la tâche sort en *time-out*,
- Pour « démarrer » la période:
  - la faire passer à l'état *active* (appel à `rate_monotonoc_period`)

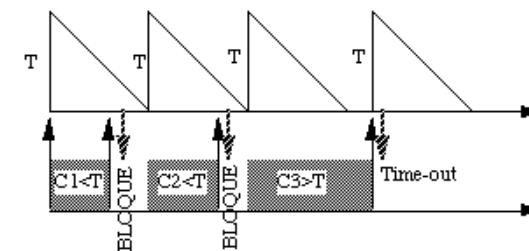
### *Implantation*

- Une interruption est générée à chaque clock\_tick :
  - elle est gérée par un ISR qui appelle une fonction RMS lorsque le nombre de clock\_ticks atteint la période
  - cette fonction réinitialise le compteur de l'ISR et change l'état de la période pour débloquer la tâche associée
- Il y a un timer par période

### *Exemple : Gestion RMS*

- Mise en œuvre d'une gestion RMS :

```
#define T 10
...
rtems_rate_monotonic_create (nom, &Periode) ;
while () {
    Etat = rtems_rate_monotonic_period(Periode, T) ;
    if (Etat == RTEMS_TIMEOUT) break ;
    /***** Code à exécuter (C1, C2, C3) *****/
}
/***** Echéance dépassée *****/
```



## Le système RTEMS

---

### **Exemple : Gestion RMS**

- `rtems_rate_monotonic_period` : initialise un PCB pour gérer la période
- Les appels à `rtems_rate_monotonic_period` :
  - Appel 1 (comportement spécifique d'initialisation) : période initialisée à T (ici 10), la tâche ressort immédiatement de la fonction,
  - Appel 2 : la tâche est bloquée pendant T- C1
  - Appel 3 : la tâche est bloquée pendant T- C2
  - Appel 4 :  $C3 > T$ , l'échéance est dépassée, la tâche ressort immédiatement avec un message d'erreur,
- Tâche critique et gestion RMS :
  - si certaines tâches sont critiques (strict respect des échéances) et d'autres, non :
    - affecter des priorités telles que celle de la tâche critique la moins prioritaire soit plus prioritaire que la tâche non-critique la plus prioritaire
    - en cas de surcharge, seules les tâches critiques respecteront leurs échéances,

## Le système RTEMS

---

### **Exemple : Fichier system.h**

- Un extrait du fichier system.h illustre ce qui est fait pour ne charger que les managers utilisés :

```
...  
  
/* configuration information */  
  
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER  
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER  
  
#define CONFIGURE_MAXIMUM_POSIX_THREADS 1  
#define CONFIGURE_MAXIMUM_POSIX_KEYS 10  
#define CONFIGURE_MAXIMUM_POSIX_MUTEXES 10  
  
#define CONFIGURE_POSIX_INIT_THREAD_TABLE  
  
#include <confdefs.h>  
...
```