Cours RT-POSIX

Etienne Borde,
Bertrand Dupouy

Pourquoi POSIX?

- Qu'est-ce que POSIX?
  - Un standard IEEE
  - Une API, et donc une librairie
  - Fournis par le Système d'Exploitation (SE)
  - Gestion des processus et threads (processus légers)
- Alors, pourquoi POSIX?
  - Portabilité: **P**ortable **O**perating **S**ystem **I**nterface
  - Une application utilisant l'API POSIX pourra être exécutée sur n'importe quel SE qui implémente POSIX (Linux, … mais aussi … INTEGRITY, LynxOS, RTEMS, VxWorks, …)

Pourquoi RT-POSIX?

- De plus en plus d'applications temps-réels
- SE classique non-adaptés à un gestion **deterministe** du temps
- Une des raison de sa popularité est d'être utilisable sur une machine Linux, puis portable sur des OS propriétaires

En bref, POSIX=API système… Va-t-on parler 1h30 d'une API? …

Plan du cours

- ***Introduction aux systèmes temps-réels d'un point de vue SE.***
- Facteurs intervenants dans la gestion du temps, toujours d'un point de vu SE
- Propositions RT-POSIX (l'API)
- Patron d'implémentation pour modèle d'ordonancement théorique

Les composants d'un SE

- Gestionnaire de fichiers
- Gestion de la mémoire
- Gestion de l'accès aux resources
  - de calcul (processeur)
  - entrées/sorties (périphériques)

Gestion du temps dans un SE? …
Ordonnaceur! ... Mais pas seulement

Le temps dans un SE classique

- Un système d'exploitation classique doit organiser et optimiser l'utilisation des ressources de façon à ce que l'accès à ces ressources soit équitable.
- Un SE n'a donc pour seule contrainte de temps que celle d'un **temps de réponse satisfaisant** pour les applications

➔ Critère de qualité: temps de réponse.

Le temps dans un SE classique

- Le SE est capable de **dater** les événements qui surviennent au cours du déroulement de l'application :
  - mise à jour d'un fichier, envoi d'un message,
  - suspension d'un traitement pendant un certain délai (sleep sous Unix), lancement d'une tâche à une certaine date.

- Mais, en aucun cas, il ne **garantit** que les résultats seront obtenus pour une **date précise**, surtout si ces résultats sont le fruit d'un traitement nécessitant des interactions avec l'environnement **physique** du système informatique.
  - Ex: **`sleep(t)`** bloque un processus pendant au moins t unités de temps …

Pourquoi d'ailleurs?

Le temps dans un SE classique

- Le SE ne prend pas en compte les contraintes d'échéances dans l'ordonnacement
- L'ordonnacement, basé sur des priorités dynamiquement attribuées, dépend du **type d'événement** attendu/reçu et de la **consommation** de temps cpu des processus (vieillissement)
- L'ordonnancement ne prend pas en compte l'échéance de réalisation de la tâche.

Rappels sur les Systèmes Temps-Réels (STR)

- Objectif des STR :
  - déterminisme temporel (en plus du déterminisme logique où les mêmes données en entrées donnent les mêmes résultats):
    - Respect des échéances, prédictibilité : répondre à des contraintes temporelles (sur le début et/ou la fin des activités),
    - Résultat correct = résultat exact ... **et fourni à la date voulue**
- Exemples :
  - Logiciel de lecture vidéo: synchronisation des flux audio/vidéo,
  - Logiciel de déclenchement d'un airbag de voiture

- Note1 : ces exemples introduisent une notion de **criticité** et de **précision des contraintes temporelles**
- Note2 : POSIX=interface de programmation seulement, donc sa capacité à être utilisé dans un domaine d'application dépend de ***l'implémentation*** de cette API et de la ***façon dont elle est utilisée.***

Synthèse des différences SE/STR

SE

- Les performances sont jugées suivant le rendement : exécuter le plus de tâches possibles, le plus rapidement possible,

C'est le SE qui décide de la dynamique d'exécution (CONTRAINTES LOGICIELLES)

STR

- Le critère de performance est le suivant : **respect de toutes ou d'une partie (en cas de surcharge) des échéances**, qu'elles soient périodiques ou non. Si on exige le respect de toutes les échéances, on parle de TR dur, sinon TR souple (mou, soft).

C'est l'environnement extérieur qui impose la dynamique d'exécution (CONTRAINTES PHYSIQUES)

Le temps dans les STR

- Real-time computing is not fast computing:

FAUX	VRAI
Information soumise à des conraintes temps-réel = information à obtenir **rapidement** Traitement temps réel = traitement à effectuer **rapidement**	Information soumise à des contraintes temps réel = information à obtenir **avant** une certaine date Traitement temps-réel = traitement à effectuer **avant** une certaine date

- Ex: Airbag…

Tâche urgente et tâche critique

- Chaque tâche a un degré :
  - d'urgence, lié à la date de son échéance;
  - de criticité, lié à son importance relative.

- Mais : une tâche très importante peu avoir de faibles contraintes de temps et tâche peu importante de fortes contraintes de temps … Pb. des ordonnancements du type RMS où le seul critère est la période.

- Comment conjuguer ces deux critères ? c'est à dire comment refléter l'urgence ET l'importance des tâches ?
  - MUF (Maximum Urgency First),
  - Systèmes paritionés (ARINC653)
  - Mixed criticality

Types d'ordonnancement TR

- Hors ligne
  - l'ordonnancement est calculé a priori, c'est à dire avant l'exécution (time driven scheduling), l'ordonnanceur se réduit à un séquenceur.
- En–ligne :
  - l'ordonnancement est décidé à l'exécution, la détection des surcharges est plus difficile… ***Quelles surcharges d'ailleurs?***
- Ne pas confondre :
  - Hors ligne / en ligne.
  - Priorité fixe / priorité dynamique.
  - Préemptif / non préemptif.
  - Priorité / criticité.

Réalisations d'applications TR

- Déterminisme temporel, il faut donc :
  - maitriser les temps d'exécution (début/fin),
  - garantir l'ordre d'exécution des fonctions (contraintes de précédence),
  - prouver l'ordonnnançabilité, donc utiliser des techniques d'ordonnancement et de gestion de la concurrence éprouvées,
  - ne pas se contenter des tests qui ne sont pas toujours assez près des conditions réelles (cf. accumulation des dérives d'horloge dans une applicationrépartie très longue)

- Sûreté de fonctionnement, il faut pouvoir :
  - Détecter les erreurs (y compris temporelles, dépassement d'échéances par exemple)
  - Confiner les erreurs, càd éviter leur propagagtion (partitionnement spatio-temporelle, voir cours sur ARINC)
  - Corriger les erreurs (cf. redondance matérielle)

Plan du cours

- Introduction aux systèmes temps-réels d'un point de vue SE.
- ***Facteurs intervenants dans la gestion du temps, toujours d'un point de vu SE***
- Propositions RT-POSIX (l'API)
- Patron d'implémentation pour modèle d'ordonancement théorique

Facteurs intervenants dans la gestion du temps

- Facteurs logiciels :
  - synchronisation (partage de ressources),
  - Algorithmes de gestion de la mémoire : (GC, pagination, caches, multi-coeurs),
  - entrées-sorties (pas de priorité),
  - gestion des disques (algorithme de parcours, allocation),
- Facteurs matériels :
  - gestion des interruptions,
  - mémoires caches,
  - pipe-line,
  - entrées-sorties matérielles,
- et encore :
  - format de l'exécutable (édition de liens statique/dynamique)
  - protocole réseau pour les SE répartis
  - la mesure doit-elle se faire dans la configuration la plus défavorable (worst case execution time, WCET) ? Mixed-Criticality ? Allowance?

Temps-réel, gestion de la mémoire, et codage

- Gestion statique :
  - le nombre, la taille et l'emplacement des objets sont connus, ou bornés, lorsque l'application est lancée,
  - avantage : accès aux objets en temps constant et faible (objets implantés sous forme de tableaux)
  - inconvénient : pas flexible
- Gestion dynamique :
  - avantage : souplesse
  - inconvénients :
    - temps d'accès et d'allocation difficile à prédire ou à borner
    - déterminisme de la désallocation
- Pratique des systèmes fortements critiques: pas d'allocation dynamique de mémoire.

Temps-réel, gestion de la mémoire, et pagination

- Temps d'accès aux informations difficile à prédire
- Ce temps d'accès peut être très long : accès disques possibles
- la pagination est donc peu utilisée en TR, ou bien avec des mécanismes de verrouillage des pages en mémoire

Temps réels et mémoires caches

- Avantage de l'utilisation de caches :
  - diminue le temps d'exécution des tâches de manière probabiliste (cf. hit ratio)

- Inconvénients :
  - augmentation du temps de changement de contexte (réinitialisation des caches) si les espaces d'adressage sont séparés, d'où utilisation de threads dans les STR
  - moins de prédictibilité, il existe diverses méthodes d'estimation du comportement des caches

- Techniques mises en œuvre : verrouillage, partage des caches

Autres éléments à prendre en compte

- Intérruptions
- Gestion des entrée/sorties

En résumé, les techniques d'accélération matériel:

Diminuent le temps moyen d'exécution d'un programme

Augmentent la dispertion des temps d'exécution (soit la différence entre BCET et WCET)

Plan du cours

- Introduction aux systèmes temps-réels d'un point de vue SE.
- Facteurs intervenants dans la gestion du temps, toujours d'un point de vu SE
- ***Propositions RT-POSIX (l'API)***
- Patron d'implémentation pour modèle d'ordonancement théorique

POSIX, l'API

- L'API POSIX permet de gérer des threads, c'est-à-dire des processus léger
  - Partagent le même espace d'adressage (celui du processus qui les crée)
  - Changement de contextes plus court
- Norme POSIX 1003.4 pour la portabilité des applications TR :
  - définit une interface standard entre l'application et le système
  - ne spécifie PAS l'implantation, mais propose des outils de mesure des performances
  - peu de fonctionnalités obligatoires

POSIX, l'API

- POSIX 4 définit la panoplie TR minimale, 4a les threads et 4b les extensions. POSIX 4b propose des outils tels que :
  - l'accès direct aux interruptions depuis les applications,
  - l'ordonnancement "serveur sporadique",
  - les ordonnancements: SCHED_FIFO, SCHED_RR, SCHED_OTHER
  - une fonction qui permet à un thread de suivre la consommation cpu d'un autre thread,
  - les files de message (mq_open, mq_receive, ...)
  - Les signaux temps réel
- Pour vérifier si la partie de la norme que l'on veut utiliser est bien implantée, utiliser ifdef et error pour être averti par le préprocesseur, ou sysconf pour un message à l'exécution:

```
#include <unistd.h>
#ifndef _POSIX_PRIORITY_SCHEDULING
#error POSIX : pas d'ordonnancement TR
#endif
```

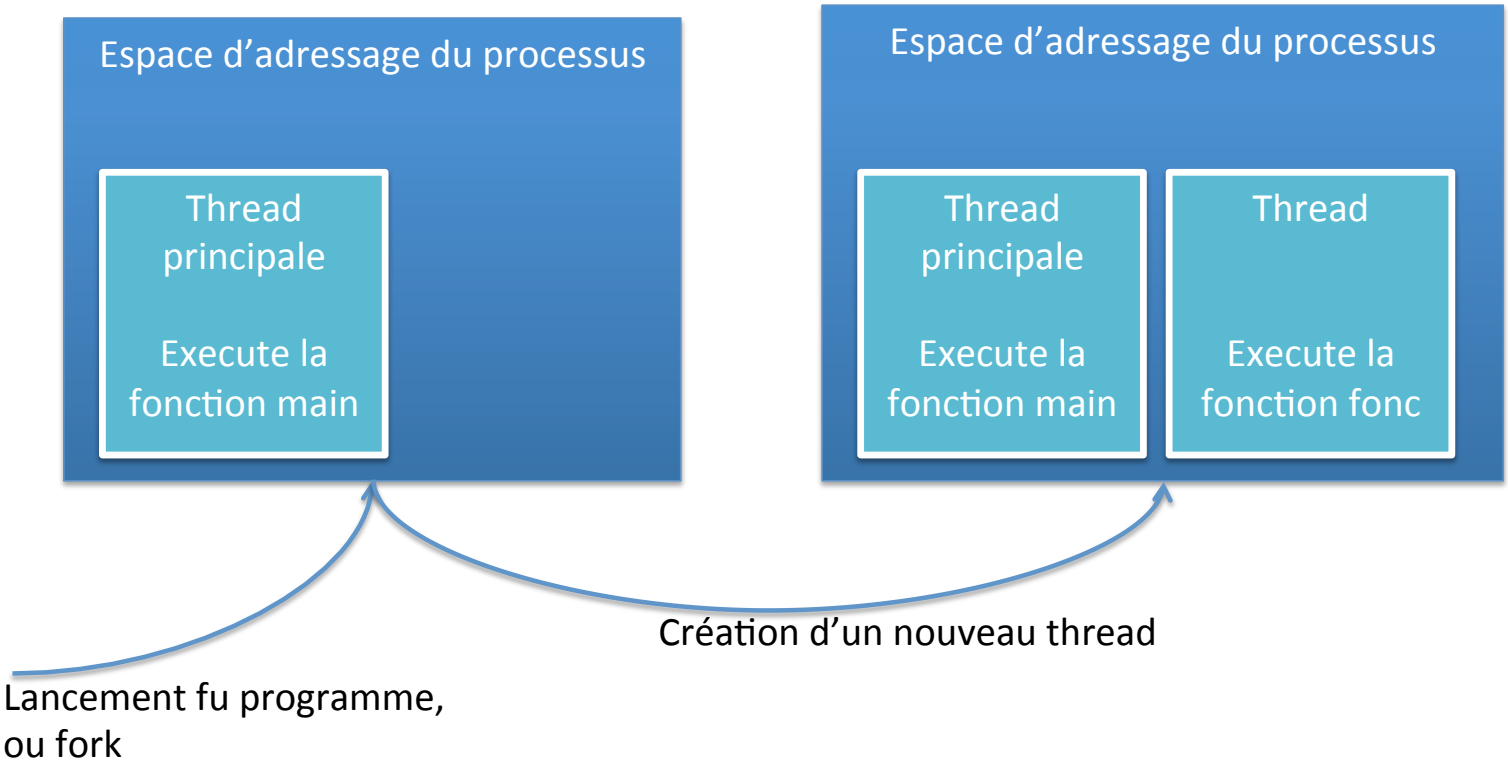
POSIX, création de threads

- Les threads peuvent être créés à partir de la librairie pthread:
  - Header: pthread.h
  - Link option for gcc: gcc –o app app.c –lpthread
- La fonction pour créer un thread est pthread_create:

```
int pthread_create(pthread_t * thread,
pthread_attr_t * attr, void *(*start_routine)(void
*), void * arg);
```

  - thread est une structure passé par adresse, et initialisée par pthread-create; on pourra la réutiliser avec d'autre fonctions de l'API
  - attr est un pointeur (peut être NULL) vers une structure pthread_attr_t. Cette structure peut être initialisé en utilisant les fonctions du type pthread_attr_*(). Elle peut être utilisé pour mettre à jour un ensemble varié de propriétés d'un thread (detach policy, scheduling policy, etc.)
  - start_routine est un pointeur de fonction qui désigne la fonction qui sera exécuté par le thread. Le thread est détruit à la fin de l'exécution de cette fonction.
  - arg est un argument passé à la fonction start_routine.

Partage de l'espace d'adressage



Exemple de code

```
#include <pthread.h>
void *thread(void *data) {
    int i;
    for (i = 0; i < 100; i++) {
        printf(« Hello world from thread »);
    }
}
int main(void) {
    pthread_t th;
    pthread_create(& th, NULL, thread, NULL);
}
```

Affichage: rien le plus souvent…

Attendre la fin d'un thread

- Quand la fonction main se termine, tous les threads créés par le processus sont détruits

- `pthread_join(pthread_t thread, void **value_ptr)` est la fonction à utiliser pour suspendre un thread (passe dans l'état bloqué) jusqu'à ce qu'un autre thread termine son exécution
- `int pthread_t thread, void **value_ptr`
  - *thread: l'identifiant du thread don't on attend la terminaison*
  - *value_ptr, recoit la valeur passé en argument de l'appel à* void pthread_exit(void **value_ptr*); dans le thread attendu.
- Les threads peuvent aussi être détachés et devenir indépendents:
  - `int pthread_detach(pthread_t thread);`

Exemple de code

```
#include <pthread.h>
void *thread(void *data) {
    int i;
    for (i = 0; i < 100; i++) {
        printf(« Hello world from thread »);
    }
}
int main(void) {
    pthread_t th;
    pthread_create(& th, NULL, thread, NULL);
    pthread_join(th, NULL);
}
```

Affichage: Hello world from thread Hello world from thread Hello world from thread Hello world from thread Hello world from thread Hello world from thread etc….

Annulation d'un thread (par l'exemple)

```
#include <pthread.h>
void *thread(void *data) {
    while(1) {
        printf(« Hello world from thread »);
    }
}
int main(void) {
    pthread_t th;
    pthread_create(& th, NULL, thread, NULL);
    sleep(1);
    pthread_cancel(& th);
    pthread_join(& th, NULL);
    return 0;
}
```

L’ordonnancement en POSIX

- Ordonnancement préemptif à priorités fixes
  - 32 niveaux de priorité doivent être proposés
- les politiques de gestion des files d'attentes associées à ces priorités sont : FIFO, RR, OTHERS… “Within priorities”
- seul l'utilisateur privilégié (root) peut accéder à ce service d'ordonnancement pour choisir FIFO ou OTHERS
- On peut aussi mettre à jour la priorité d’un thread via ses attributs (voir exemple ci-après)

Note: toutes ces fonctions sont applicables qu’aux threads; quand on peut appliquer des fonctions POSIX aux processus, le nom de ces fonction ne contient pas le mnémonique pthread

Exemple: mise à jour de la priorité d'un thread

```
#include <sched.h>
pthread_t tid;
pthread_attr_t attr;
struct sched_param param;
...
pthread_attr_init (&attr)

/******** politique d'ordonnancement ********/
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);

/******** priorité du thread ********/
param.sched_priority = 1;
pthread_attr_setschedparam (&attr, &param);
/******** création du thread ********/
pthread_create (&tid, &attr, fonc, NULL);
```

POSIX et ordonnancement
Divers

- Connaitre les niveaux de priorité authorisés (dépend de l'implémentation; 32 niveaux min.)
  - prio_max = `sched_get_priority_max(policy);`
  - prio_min = `sched_get_priority_min(policy);`

- Connaitre la valeur du quantum pour la politique RR (pour le processus courant)
  - `struct timespec qtm;`
  - `sched_rr_get_interval(0,&qtm);`

Mutexes

- Les mutex sont destinés à la gestion des accès aux sections critiques (exclusion mutuelle)
  - la file d'attente qui leur est associée est gérée par ordre de priorités décroissantes, ils peuvent être utilisés entre threads ou processus, suivant les options :
  - `pthread_mutex_init()`, `pthread_mutex_lock()`, `pthread_mutex_trylock()`, `pthread_mutex_unlock()`, `pthread_mutexattr_*`, …

Exemple de code

```
//variable globale
pthread_mutex_t lock;
...
// initialisation du STR (i.e. avant la création des threads)
pthread_mutex_init(& lock, NULL);
... // création des threads
```

```
// dans les threads concurrents
ret = pthread_mutex_lock(& lock);
… // section critique
ret = pthread_mutex_lock(& lock);
```

```
// dans les threads concurrents
ret = pthread_mutex_lock(& lock);
… // section critique
ret = pthread_mutex_lock(& lock);
```

```
// à la fin du processus
pthread_mutex_destroy(& lock);
```

Attributs des mutex

- Le second argument de `pthread_mutex_init()` est un ensemble d'attributs spécifiques aux mutexes, et regroupés dans une structure `pthread_mutexattr_t`. une telle structure peut-être initialisée et manipulé avec des fonctions du type `pthread_mutexattr_*()`

- Exemple:

```
int pthread_mutexattr_settype
(pthread_mutexattr_t *attr, int type);
```

où type peut prendre comme valeur: PTHREAD_MUTEX_NORMAL, PTHREAD_MUTEX_ERRORCHECK, PTHREAD_MUTEX_RECURSIVE, PTHREAD_MUTEX_DEFAULT

Variables conditionnelles

- Les variables conditionnelles permettent de suspendre l'exécution d'un thread jusqu'à ce qu'une condition devienne vrai; cette condition est signalée par un autre thread.

- Initialisation
  - pthread_cond_t cond;
  - pthread_cond_init(& cond, NULL);
- Attente:
  - pthread_cond_wait(& cond, & mutex)
  - **Toujours bloquant**
  - Le mutex passé en paramètre est sera libéré avant la mise en attente (de façon atomique), puis **repris immédiatement au réveil** (trylock)
- Signalisation:
  - A un thread en attente (pas forcément FIFO): pthread_cond_signal(& cond);
  - A tous les threads en attente: pthread_cond_broadcast(& cond);
  - **Non mémorisé** (perdu si aucun thread en attente)

Example de code

```
/****** variables partagees ******/
pthread_mutex_init(&Verrou, NULL);
pthread_cond_init(&VarCond, NULL);
// création de threads
```

```
...
while (...){
  ...
  pthread_mutex_lock(&Verrou);
  Compteur ++;
  if (Compteur > N)
    pthread_cond_broadcast(&VarCond);
  pthread_mutex_unlock (&Verrou);
  ...
}
```

Vérfifie qu'un seuil est atteint

```
...
pthread_mutex_lock (&Verrou);
while (N < Compteur) {
    pthread_cond_wait(&VarCond,
&Verrou);
}
printf ("Seuil atteint! "\n);
...
pthread_mutex_unlock (&Verrou);
```

Vérfifient que le seuil soit atteint

Timers

Avec l'option CLOCK_REALTIME :

- ✧ `clock_settime`: Initialiser l'horloge
- ✧ `clock_gettime`: Lire la valeur de l'horloge
- ✧ `clock_getres`: Lire la résolution de l'horloge
- ✧ `nanosleep`: Sleep haute résolution
- ✧ `timer_create`: Création d'un timer
- ✧ `timer_delete`: Destruction d'un timer
- ✧ `timer_settime`: Armement/désarmement d'un timer
- ✧ `timer_gettime`: Lire le délai restant sur un timer
- ✧ `timer_getoverrun`: Lire le délai dépassé sur un timer

Signaux

- Dans l'implémentation TR les différentes occurrences d'un même signal sont **conservées**, le nombre de signaux reçus correspond toujours au nombre de signaux émis.
  - Pas de perte : gestion d'une liste de signaux en attente
  - La priorité liée au signal est respectée (celle du thread) dans la gestion de la file d'attente
- Emission par sigqueue, par un timer, par un fin d'e/s
- nouveaux signaux : RTSIG_MAX signaux, numérotés de SIGRTMIN à SIGRTMAX

✧ `sigqueue`: Mettre un signal dans la file d'attente associée au processus destinataire

✧ `sigwaitinfo`: Attendre un signal et une info

✧ `sigtimedwait`: Attendre d'un signal avec temporisation

Sémaphores

Les sémaphores sont l'implantation classique de l'outil défini par Dijkstra. La file d'attente est gérée par ordre de priorités décroissantes, les sémaphores peuvent être utilisés entre threads ou processus, suivant les options.

- ✧ `sem_open`: open and / or create a named semaphore.
- ✧ `sem_close`: close a named semaphore
- ✧ `sem_unlink`: destroy a named semaphore
- ✧ `sem_init`: initialize an unnamed semaphore
- ✧ `sem_destroy`: destroy an unnamed semaphore
- ✧ `sem_getvalue`: get current semaphore count
- ✧ `sem_wait`: Try to lock the semaphore. Wait otherwise.
- ✧ `sem_trywait`: Just tries to lock the semaphore, but gives up if the semaphore is already locked.
- ✧ `sem_post`: Release the semaphore.

Files de messages

Elles sont similaires à ceux proposés par les IPC System V, mais à chaque message est associée une priorité. Le problème de l'inversion de priorité n'est pas géré :

- ✧ `mq_close`: fermer une file de messages
- ✧ `mq_getattr`: récupérér 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

Plan du cours

- Introduction aux systèmes temps-réels d'un point de vue SE.
- Facteurs intervenants dans la gestion du temps, toujours d'un point de vu SE
- Propositions RT-POSIX (l'API)
- ***Patron d'implémentation pour modèle d'ordonancement théorique***

Ordonnancement RMS

```
/*
 * scenario:
 *      - 2 periodic threads T1 (period=1000 ms) and T2 (period=2000 ms);
 *      - 1 sporadic thread T3 (period=3000 ms);
 *      - 1 global variable gv (integer); protected with PCP;
 *      - T1 increments gv; T2 and T3 displays gv;
 *      - Scheduling policy is RMS.
 */
```

Ordonnancement RMS

```
int main()
{
...
  pthread_create(&tid1, &attr1, (void* (*)(void*))body_of_T1, NULL);
  pthread_create(&tid2, &attr2, (void* (*)(void*))body_of_T2, NULL);
  pthread_create(&tid3, &attr3, (void* (*)(void*))body_of_T3, NULL);
}

void body_of_T1()
{…}
void body_of_T2()
{…}
void body_of_T3()
{…}
```

Que manque-t-il?

Ordonnancement RMS

```
int main()
{
...
  pthread_create(&tid1, &attr1, (void* (*)(void*))body_of_T1, NULL);
  pthread_create(&tid2, &attr2, (void* (*)(void*))body_of_T2, NULL);
  pthread_create(&tid3, &attr3, (void* (*)(void*))body_of_T3, NULL);

  // wait for threads to finish (otherwise the process terminates
  // immediately)
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  pthread_join(tid3, NULL);
}

void body_of_T1()
{…}
void body_of_T2()
{…}
void body_of_T3()
{…}
```

Ordonnancement RMS

```
void body_of_T1()
{
  unsigned int iter=0;
  while(1)
    {
      iter++;
      printf("Executing T1 iter %d\n", iter);
      // Compute next dispatch time
      clock_gettime(CLOCK_REALTIME, &T1_timer);
      T1_timer.tv_sec = T1_timer.tv_sec+iter*PERIODT1_s;
      T1_timer.tv_nsec = T1_timer.tv_nsec;

      // Wait for next dispatch time
      pthread_mutex_lock (&T1_mutex);
      pthread_cond_timedwait (&T1_cond, &T1_mutex, &T1_timer);
      pthread_mutex_unlock (&T1_mutex);


    }
}
```

Problème? …

Départ synchronisé des threads

- Attendre dans le main que tous les threads soient synchro
- Attendre dans les threads que le main donne le signal (pas forcément signal au sens SE) de départ
- Utiliser des dates de réveil absolues

Attendre dans le main que tous les threads soient synchro

```
int main()
{
  ...
  pthread_create(&tid3, &attr3, (void* (*)(void*))body_of_T3, NULL);
  // wait a bit for the end of the threads creation before to release them;
  sleep(2);
  clock_gettime(CLOCK_REALTIME, &init_time);
  pthread_cond_broadcast(&threads_init_cond);
  …
}

void body_of_T1()
{
  // wait all threads have been created and initialized
  pthread_mutex_init(&T1_mutex, NULL);
  pthread_cond_init (&T1_cond, NULL);
  pthread_mutex_lock(&threads_init_mutex);
  pthread_cond_wait(&threads_init_cond, &threads_init_mutex);
  pthread_mutex_unlock(&threads_init_mutex);
  …
}
```

Utiliser des dates de réveil absolues

```
void body_of_T1()
{
  ...
  unsigned int iter=0;
  while(1)
    {
      iter++;
      T1_timer.tv_sec = init_time.tv_sec+iter*PERIODT1_s;
      T1_timer.tv_nsec = init_time.tv_nsec;

      ...

      pthread_mutex_lock (&T1_mutex);
      pthread_cond_timedwait (&T1_cond, &T1_mutex, &T1_timer);
      pthread_mutex_unlock (&T1_mutex);
    }
}
```

Notes à propos de la solutions

- Devrait être généralisée pour un ensemble de N threads, en fournissant une API de plus haut niveau: Middleware.

- Ce n'est qu'une solution possible, il en existe d'autres, peut-être mieux…

Synchro PCP

- Directement fourni par POSIX

```
pthread_mutex_t mutex;
pthread_mutexattr_t mutex_attr;

int main()
{
  pthread_mutexattr_setprotocol(&mutex_attr, PTHREAD_PRIO_PROTECT);
  pthread_mutexattr_setprioceiling(&mutex_attr, PRIOT1);
  pthread_mutex_init(&mutex, &mutex_attr);
  ...
}
```

Sporadic server

- Directement fourni par POSIX:
  - Init

```
int main()
{
  /****** Initialisation des priorites ******/
  /****** Initialisations de la periode et du budget ******/
  /****** a 1/2 seconde et 1/4 seconde ******/
  #define HIGH_PRIORITY 150
  #define LOW_PRIORITY 100
  schedparam.ss_replenish_period.tv_nsec = 500000000;
  schedparam.ss_initial_budget.tv_nsec = 250000000;
  schedparam.sched_priority = HIGH_PRIORITY;
  schedparam.ss_low_priority = LOW_PRIORITY;
  …
}
```

Sporadic server

Dans le thread serveur sporadic, on simule le comportement (1/3):

```
/********
Boucle pour voir diminuer la priorite
**********/
for ( ; ; ) {
        if ( schedparam.sched_priority != LOW_PRIORITY )
                continue;
        priority = schedparam.sched_priority;
        sprintf( buffer, "-nouvelle priorite = %d", priority );
        print_current_time( buffer );
        /**********
        L'appel a lock va augmenter la priorite
        **********/
        puts( "Verrou va etre pris" );
        pthread_mutex_lock( &mutex );
        priority = schedparam.sched_priority;
        sprintf( buffer, "-nouvelle priorite = %d", priority );
        print_current_time( buffer );
        break;
}
```

Sporadic server

Dans le thread serveur sporadic, on simule le comportement (2/3):

```
/*****
Attendre pour voir le budget etre re-alimente
**********/
for ( ; ; ) {
        if (schedparam.sched_priority == HIGH_PRIORITY )
                break
}

priority = schedparam.sched_priority;
sprintf( buffer, "-nouvelle priorite = %d", priority );
print_current_time( buffer );
```

Sporadic server

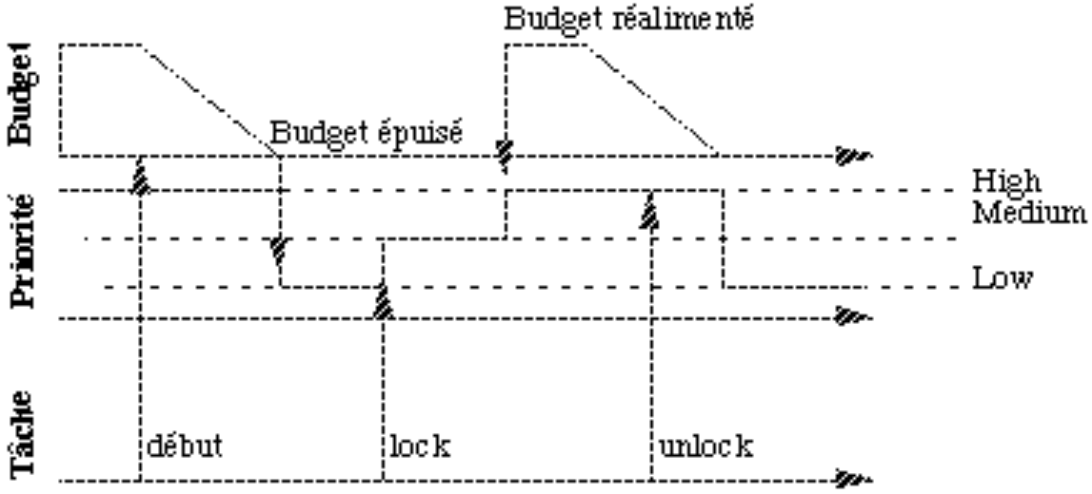
Dans le thread serveur sporadic, on simule le comportement (3/3):

```
/*****
Le unlock doit faire descendre la priorite
**********/

puts( " On va rendre le verrou" );
pthread_mutex_unlock( &mutex );
priority = schedparam.sched_priority;
sprintf( buffer, "-nouvelle priorite = %d", priority );
print_current_time( buffer );
for ( ; ; ) {
        if ( schedparam.sched_priority == LOW_PRIORITY )
                break;
}
priority = schedparam.sched_priority;
sprintf( buffer, "-nouvelle priorite = %d", priority );
print_current_time( buffer );
```

Résultat

```
Fri May 24 11:05:01 - nouvelle priorite = 150
Fri May 24 11:05:01 - nouvelle priorite = 100 Verrou va etre pris
Fri May 24 11:05:01 - nouvelle priorite = 131
Fri May 24 11:05:01 - nouvelle priorite = 150 On va rendre le verrou
Fri May 24 11:05:01 - nouvelle priorite = 150
Fri May 24 11:05:01 - nouvelle priorite = 100
```



Conclusion

- API RT-POSIX riche, très utilisée en pratique dans les STR, pour sa portabilité
- API bas niveau, don't l'usage mérite être factorisé via un middleware
- Attention à la correspondance entre modèle d'ordo théorique et implémentations
- Autres standard d'OS temps-réel existent: OSEK, ARINC653…