Master 1 - Informatique

UE *NSTR*

Noyaux de Systèmes Temps Réel

2013-2016

Zoubir Mammeri

Chapitre 1

Introduction aux systèmes temps réel

1. Définitions et concepts de base

La gestion du temps dans un OS classique (temps partagé)

- Un système d'exploitation (OS) classique, c.-à-d. orienté temps partagé, doit organiser et optimiser l'utilisation des ressources de façon à ce que l'accès à ces ressources soit **équitable**. Un OS n'a donc pour seule contrainte de temps que celle d'un temps de réponse satisfaisant pour les utilisateurs avec lesquels il dialogue. Les traitements qu'on lui soumet sont effectués en parallèle au fil de l'allocation des quanta de temps aux diverses applications.

- Un OS 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, etc. Il permet aussi de suspendre un traitement pendant un certain délai, d'en lancer un à une certaine date...

- En aucun cas, un OS classique ne garantit que les résultats seront obtenus pour une date précise, surtout si ces résultats sont le fruit d'un traitement déclenché par un événement EXTERIEUR (émission d'un signal par un périphérique quelconque,...)

Définitions de systèmes temps réel

- « Intervalle de temps compatible avec le rythme réel d'arrivée des données et à l'intérieur duquel un ordinateur peut effectuer les traitements nécessaires » [Le petit Robert].

- « Temps réel signifie l'aptitude d'un système d'exploitation de fournir le niveau de service requis au bout d'un temps de réponse borné ». [Posix 1003.1b]

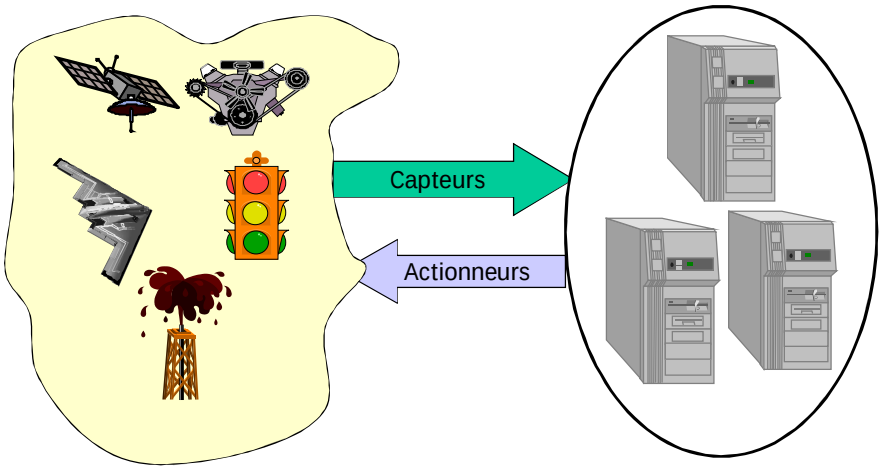
- Un système temps réel est un système informatique qui doit répondre à des stimuli fournis par un environnement externe afin de le contrôler.

- Système qui interagit de manière prédictible avec son environnement.

- Un STR est un système dont la correction dépend non seulement de la justesse des calculs mais aussi du temps auquel est fourni la réponse (contraintes temporelles). En d'autres termes, un résultat hors délai et un résultat faux.

Attention : Ne pas confondre temps réel et rapidité
Temps réel veut dire prédictibilité et non rapidité.

Systèmes temps réel typiques

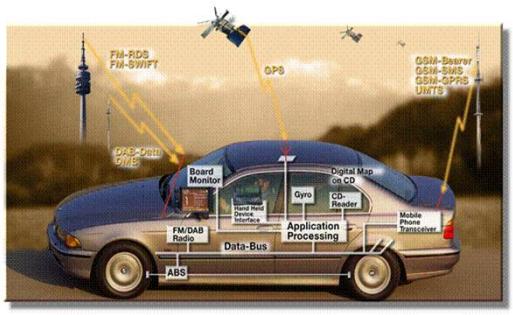


Domaines d'application des STR

- Domaines « Installations lourdes »
  - installations industrielles
  - installations nucléaires
  - installations de commande/supervision (défense, aérien, espace, trafic urbain, routier...)
- Domaines « systèmes embarqués »
  - Transport : Automobile, train...
  - Robotique
  - Engins militaires
- Domaines « grand public »
  - Jeux et loisirs
  - Téléphonie, Internet mobile
  - Implants (santé, sécurité...)
  - Domotique, immeubles intelligents
  - Vêtements intelligents
  - ...

Diversités des applications TR&E

Exemples typiques de STR embarqués grand public



Définitions de systèmes embarqués

1. Un système embarqué (SE) est un système informatisé spécialisé qui constitue une partie intégrante d'un système plus large ou une machine. Typiquement, c'est un système sur un seul processeur et dont les programmes sont stockés en ROM. A priori, tous les systèmes qui ont des interfaces digitales (i.e. montre, caméra, voiture…) peuvent être considérés comme des SE. Certains SE ont un système d'exploitation et d'autres non car toute leur logique peut être implantée en un seul programme.

2. Un système embarqué est une combinaison de logiciel et matériel, avec des capacités fixes ou programmables, qui est spécialement conçu pour un type d'application particulier. Les distributeurs automatiques de boissons, les automobiles, les équipements médicaux, les caméras, les avions, les jouets, les téléphones portables et les PDA sont des exemples de systèmes qui abritent des SE. Les SE programmables sont dotés d'interfaces de programmation et leur programmation est une activité spécialisée.

'Embedded' : Enfoui / Embarqué

Classification des STRE

- **Temps réel dur ('hard real-time') : le non respect des contraintes temporelles entraîne la faute du système**
  – e.g. contrôle de trafic aérien, système de conduite de missile, …

- **Temps réel souple ('soft real-time') : le respect des échéances est important mais le non respect des échéance ne peut occasionner de graves conséquences**
  – e.g. système d'acquisition de données pour affichage

- **Temps réel ferme ('firm real-time') : temps réel souple, mais si l'échéance est dépassée le résultat obtenu n'a plus de valeur (et est donc écarté)**
  – e.g. projection vidéo

Appli TR, STR

- **Application temps réel = Ensemble de tâches qui coopèrent**
- **Les tâches accèdent à des ressources communes**
- **Les tâches s'échangent des messages via un réseau**
- **Les tâches s'exécutent sur 1 ou n processeurs**
- **Système STRE vs Application STRE**
  - **STRE = Matériel + RTOS + logiciel applicatif**
  - **Application = Logiciel applicatif**
  - Attention : Les deux notions sont parfois confondues

2. Caractéristiques des STRE

- **Prédictibilité (répondre dans des temps connus)**
- **Robustesse et disponibilité**
- **Sécurité**
- **Communication, répartition**
- **Caractéristiques des systèmes embarqués**
  - ◇ Encombrement mémoire (mémoire limitée, pas de disque en général)
  - ◇ Consommation d'énergie (batterie : point faible des SE)
  - ◇ Poids et volume
  - ◇ Autonomie
  - ◇ Mobilité
- **Maintenabilité**
- **Coût de produit en relation avec le secteur cible**

3. Compétences pour la conception et développement

- Domaine métier : médical, loisirs, transport…
- Ingénierie : électricité, électronique, chimie, mécanique, robotique, …, informatique
- Sciences : mathématiques, statistiques, probabilités, recherche opérationnelle

- Informatique
  - Algorithmique
  - Programmation
  - Architectures (CPU, mémoires, périphériques)
  - Gestion de l'énergie des processeurs et périphériques
  - Génie logiciel, co-design, méthodes formelles, automates…
  - Evaluation de performances, test, simulation, vérification
  - Systèmes d'exploitation
  - Sécurité et robustesse
  - Réseaux, mobilité
  - Capteurs et actionneurs
  - Vision par ordinateur, caméra

3. Compétences pour la conception et développement

Langage de programmation

- Besoins
  - Expression des contraintes de temps
  - Expression et gestion du parallélisme
  - spécification et gestion de périphérique de bas niveau (E/S de base)
  - Modularité
  - Interfaces avec d'autres langages

- 3 familles de langages
  - Langages assembleurs
  - Langages séquentiels liés à des librairies système
  - Langages concurrents de haut niveau

Langage de programmation

- Langages assembleurs

– Historiquement, ces langages furent longtemps les seuls à être utilisés dans le contexte des STR

– Dépendant par nature de l'architecture cible (matériel et système d'exploitation)

– Aucune abstraction possible et grande difficulté de développement, de maintenance et l'évolution

– Langages à proscrire sauf pour l'implémentation de petites fonctionnalités très spécifiques et apportant une grande amélioration des performances

- Langages séquentiels liés à des librairies système

– Les plus connus sont C, C++

– Apportent un plus grand pouvoir d'abstraction et une certaine indépendance du matériel

– Mais, doivent faire appel à des librairies systèmes spécifiques pour la manipulations des processus

– Ils posent le problème de la standardisation des appels systèmes mais sont quelques fois le seul choix possible à cause de la spécificité d'une cible et des outils de développement sur celle-ci.

Langage de programmation

- Langages concurrents de haut niveau

– Langages généralistes incluant de plus la notion de tâches et des primitives de synchronisation

– Haut pouvoir d'abstraction, indépendance des architecture et des systèmes cibles

– Parmi ces langages, Ada et Java sont des plus aboutis

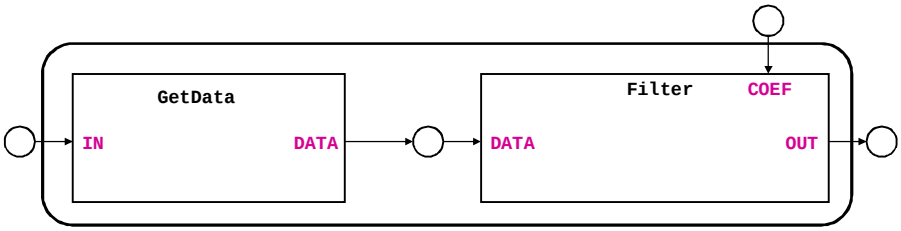
– Ces langages sont à privilégier lorsque d'autres contraintes ne rendent pas leur choix impossible.

– Langages synchrones (Lustre, Esterel...) : déterminisme garanti

– Langages asynchrones (SDL, UML, Java...) : difficulté de garantir le déterminisme

Exemple de tâches temps réel

(en langage FlowC)



La tâche `GetData` mesure une température à partir de l'environnement et envoie la valeur mesurée à la tâche `Filter`.

Une mesure est effectuée toutes les 10 ms.

Quand `N` mesures ont été effectuées, la moyenne des `N` mesures est envoyée à `Filter`.

La tâche `Filter` lit `N` mesures et les ignore et lit ensuite la moyenne des mesures, multiplie cette moyenne par c (dont la valeur est lue à partir de `COEF`) et envoie la valeur obtenue à l'environnement via le port `OUT`.

Exemple de tâches temps réel

(en langage FlowC)

```
Process GetData
(InPort IN, OutPort DATA)
{ float mesure, somme; int i;
  While(1)
  { somme = 0;
    for (i=0; i< N; i++) {
         READ(IN, mesure ,1);
         somme+= mesure ;
         WRITE(DATA, mesure, 1)
         DELAY(10)
                          }
   WRITE(DATA, somme/N,1)
}}
```

```
Process Filter
(InPort DATA, InPort COEF, OutPort OUT)
{ float c, d; int j;
  c = 1; j = 0;
  While(1)
  { SELECT(DATA, COEF) {
    CASE DATA: READ(DATA, d, 1);
         if (j==N) {j=0; d=d*c;
                   WRITE(OUT, d,1); }
         else j++;
   CASE COEF: READ(COEF, c, 1); break ;
}}}
```

4. Tendances du marché

- **Logiciel**
  - Beaucoup de fournisseurs
  - Pas de fournisseur dominant
  - Problèmes de compatibilité : Posix 1003.1c et 1b

- **Matériel**
  - Processeurs essentiellement Motorola
  - Cartes à puce
  - Périphériques d'E/S (mesures…) : très diversifiés

5. Exemple simple et éclairant

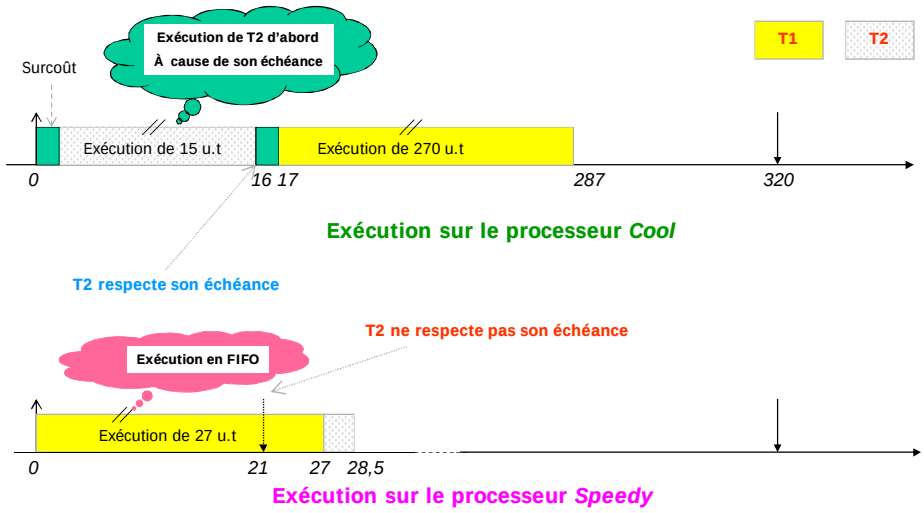
- **On dispose de deux systèmes :**
  - **Système *Cool*, avec :** processeur à vitesse 1 ; surcoût système de 1 ; ordonnancement EDF (Earliest Deadline First)
  - **Système *Speedy*, avec :** processeur à vitesse 10 ; surcoût système de 0 ; ordonnancement à l'ancienneté (FIFO)
- **Donc *Speedy* est 10 fois plus rapide et infiniment plus efficace que *Cool***

- **On considère une application composée de deux tâches non préemptibles :**
  - Tâche *T1* envoie des commandes périodiques pour entretenir la commande moteur du véhicule sa durée est de 270 sur *Cool* et de 27 sur *Speedy* ; son délai critique de 320
  - Tâche *T2* réagit à une commande du volant ; sa durée est de 15 sur *Cool* et de 1,5 sur *Speedy* ; son délai critique est de 21.

- **A l'instant *t=0*, la tâche *T1* est déclenchée pour son exécution périodique. Cependant, au même instant, *T2* est demandée en réaction à un événement.**

Les tâches *T1* et *T2* sont de même importance pour la conduite du véhicule

5. Exemple simple et éclairant



6. Sources des contraintes de temps

➔ **Origines externes**

- **Caractéristiques physiques (vitesse, durée de réaction chimique, ...)**
- **Caractéristiques liées aux lois de commande du système physique**
- **Qualité de service requise en terme de délai de réponse tolérable (qualité audio/vidéo)**
- **Perception sensorielle et le temps de réaction de l'homme**
- **Contraintes à caractère commercial**
- **Autres**

➔ **Connaissance des aspects physiques nécessaire**

6. Sources des contraintes de temps

➔ **Origines internes**

– **Choix de conception**

- architecture centralisée ou répartie (données, traitements, contrôle)
- actions périodiques (avec les périodes adéquates) ou apériodiques
- choix d'une structure d'application (interface entre composants, ...)
- autres

– **Choix d'architecture matérielle et logicielle**

- processeurs avec des vitesses particulières
- système d'exploitation (intégrant des algorithmes d'ordonnancement)
- réseau avec des débits et des temps de réponse donnés
- autres

7. Expression des contraintes de temps

Expression à l'aide du temps quantifié (temps physique)

➔ **Temps absolu**

– un instant (date)

– une fenêtre temporelle (intervalle de temps)

➔ **Temps relatif**

– une durée (délai, période), minimale, maximale, moyenne

7. Expression des contraintes de temps

Expression sur les événements

- Contraintes sur la durée de vie d'un événement
- Contraintes sur les instants et l'ordre d'occurrence d'événements

Expression sur les données

- Durée de validité (ou contrainte de fraîcheur)
- Contraintes de production
- Contraintes de corrélation

7. Expression des contraintes de temps

Expression sur les actions (opérations)

- Période
- Instant au plus tôt/tard, pour démarrer une action
- Instant au plus tard/tôt, pour finir une action
- Temps maximum d'exécution (ou échéance relative)
- Temps maximal d'attente d'une ressource
- Temps minimal/maximal d'utilisation d'une ressource
- Temps minimum d'exécution affecté à une action
- Précédence entre actions

Quelques ordres de grandeur

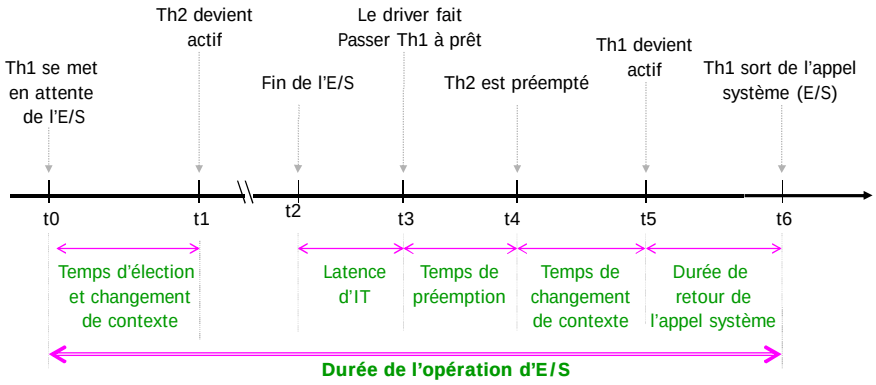
Domaines d'applications	Ordre de grandeur des CT
➔ Processus chimique	en heure
➔ Fabrication	en minute
➔ Robotique	en dizaines ms
➔ Systèmes vocaux	en ms
➔ Systèmes radar	en ms

8. Eléments constituant le temps de réponse d'une tâche

- **Temps d'exécution (Worst Case Execution Time)**
- **Temps d'attente de ressources**
- **Temps d'attente des E/S**
- **Temps dus au système d'exploitation et difficilement calculables**
  - Temps de latence des interruptions
    (délai entre l'arrivée d'une IT et le début de sa prise en compte effective)
  - Temps de préemption
    (délai de sélection de la tâche à exécuter)
  - Latence de l'ordonnancement
    (délai entre la réception de l'IT et l'entrée dans la tâche de l'utilisateur)
  - Temps de commutation de contexte de tâche
  - Latence de sémaphore
    (délai entre la libération d'un sémaphore et la réactivation de la tâche bloquée derrière ce sémaphore)

Exemple de temps de réponse à un événement

- On considère deux threads Th1 et Th2. Th1 est le plus prioritaire. A l'instant t0, Th1 lance une E/S. Le dessin ci-dessous schématise la suite des événements :



Chapitre 2

Introduction aux RTOS (systèmes d'exploitation temps réel)

1. Fonctions et mécanismes de base

Programmation temps réel =

programmation concurrente + prise en compte de contraintes de temps

- ♦ Lire le temps (horloge)
- ♦ Fixer/modifier la priorité de tâches
- ♦ Supprimer certains aspects du langage pour le rendre déterministe
- ♦ Exécuter les tâches en respectant les contraintes de temps

1. Fonctions et mécanismes de base

- Une application TRE = ensemble d'activités concurrentes éventuellement réparties

- Programmation concurrente permet
  - Expression et manipulation (création, destruction...) d'entités concurrentes (appelées tâches, processus ou threads)
  - Mise en place de moyens de communication et synchronisation entre tâches
  - Abstraction de l'exécution réelle des tâches sur une architecture mono ou multi processeur

1. Fonctions et mécanismes de base

- Processus vs tâche
  - Un processus possède son propre espace d'adressage. Il s'exécute sur sa propre machine virtuelle.
  - Le système d'exploitation met en place les mécanismes de protection inter processus.
  - Le partage de mémoire entre processus est soit impossible, soit fait de manière explicite par des appels système.
  - Une tâche partage son espace d'adressage avec d'autres tâches (mais ce n'est pas obligatoire).
  - Le programmeur doit mettre en place manuellement les mécanismes de protection inter tâches.
- Processus vs tâche vs thread
  - En Ada on parle de ***tâche***
  - En Java et dans la norme Posix, on parle de ***Process*** et ***thread***.
  - Un *process* Posix contient un ou plusieurs threads.

1. Fonctions et mécanismes de base

- Tâches indépendantes
  - Deux tâches sont indépendantes si l'exécution de l'une n'influe pas sur l'autre.
- Tâches en coopération
  - Deux tâches sont en coopération si elles mènent un travail commun nécessitant synchronisation et/ou communication.
- Tâches en compétition
  - Deux tâches sont en compétition si accèdent à des ressources partagées en exclusion.

Déclaration et démarrage d'une tâche

- Quatre possibilités
  - Déclaration explicite de tâches grâce aux constructeurs du langage (Ada, Java...)
  - Utilisation des primitives ***fork*** et ***join***
    - * ***fork*** crée un processus qui s'exécute en // avec son père
    - * ***join*** permet à un père d'attendre la fin d'exécution de son fils.
  - Utilisation de blocs ***cobegin*** et ***parbegin***
  - Utilisation de coroutines

- **Remarque**

Lorsqu'on veut programmer des tâches qui constituent une application, il faut éviter (si possible) d'utiliser *fork et* join. Il faut utiliser les primitives de création et de manipulation de threads.

Exemple de tâche avec *fork*

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#define N 50
int main(void)
  { pid_t tab_pid[N]; // pid des fils
    int status, i;
    for (i=0; i<N; i++){
      if ( ! (tab_pid[i] = fork()) ){
         // le code du fils; ici un simple affichage et un repos de une seconde
                  printf("hello\n");
                  sleep(1);
                  exit();
                                    }
   }
// Seul le père arrive ici car les 50 fils ont terminé par exit ;
// Il attend leur terminaison
   for (i=0; i<N; i++){
     waitpid( tab_pid[i], &status, 0);
                        }
    return 0;
}
```

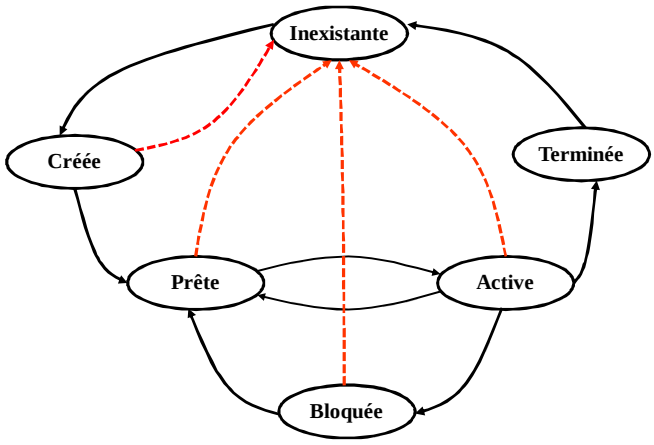
Exemple de tâche en Ada

```
task type Genre_De_Tache                    -- Spécification du type de tâche
     (param Integer);

task body Genre_De_Tache is                 -- Corps du type
        begin
          loop
                  Put(Param);
          end loop;
end Un_Genre_De_Tache;

Tache1 : Genre_De_Tache(1);                 -- Déclaration de deux tâches avec le
Tache2 : Genre_De_Tache(12);                -- type défini précédemment
```


Etats d'une tâche



Structure simplifiée d'un système

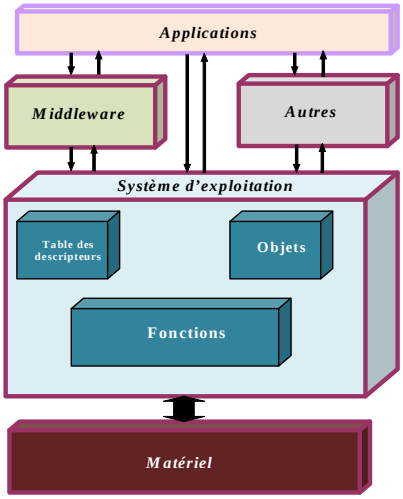
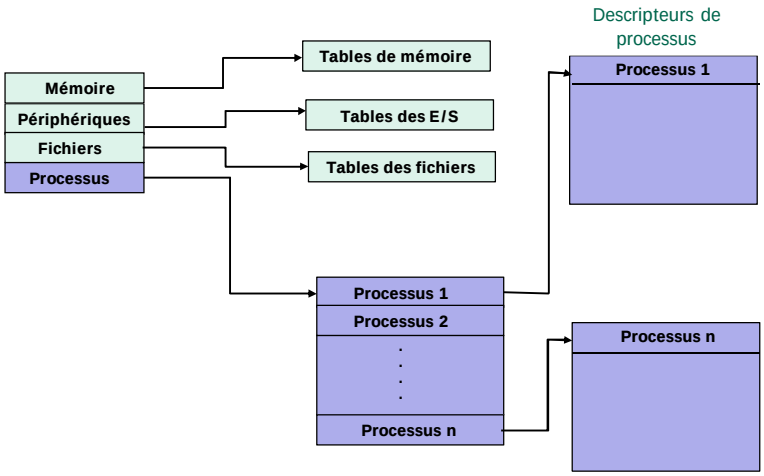
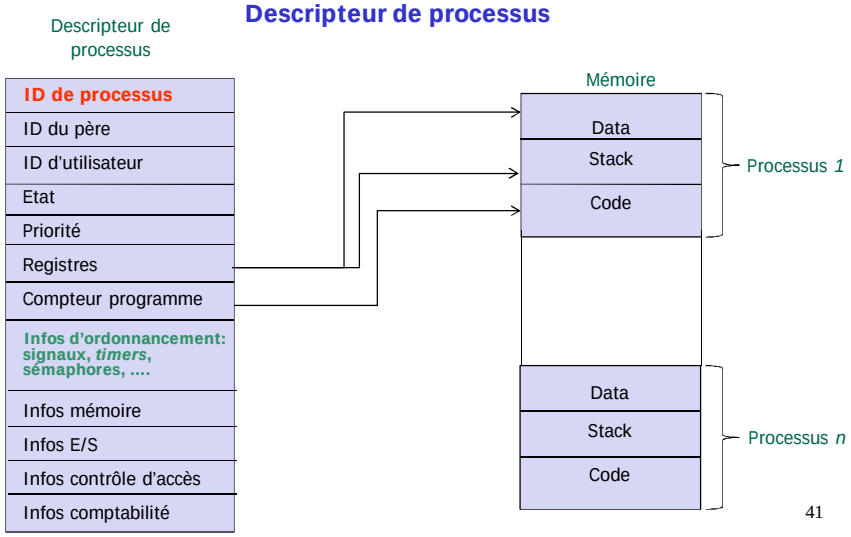


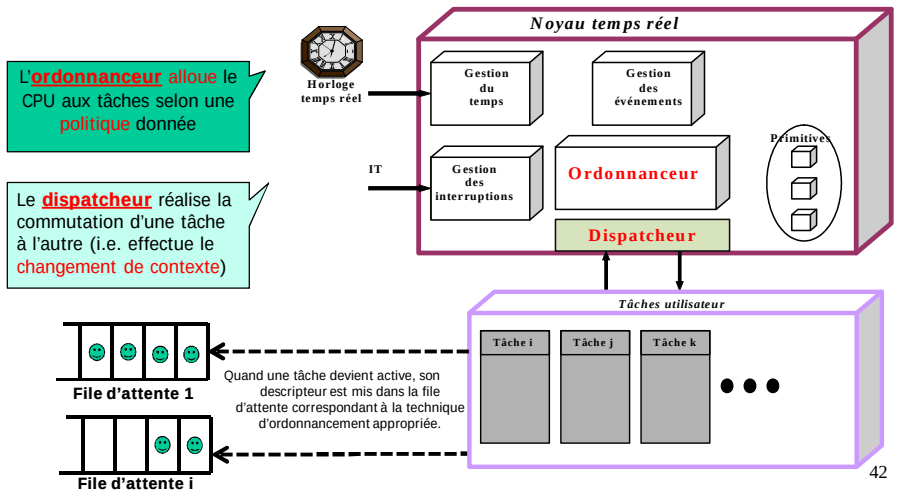
Table des descripteurs



Descripteur de processus



Structure simplifiée d'un système Temps réel



➔ L'ordonnanceur peut être : préemptif ou non préemptif

➔ L'ordonnanceur prend la main (s'exécute)

- À l'initialisation (déterminer la première tâche à exécuter)
- Quand la tâche en cours se termine (i.e. pour changer de tâche)
- Quand la tâche en cours se bloque à cause d'une ressource occupée
- Quand le quantum de temps de la tâche en cours est fini (cas du temps partagé)
- Quand une nouvelle tâche devient prête (ce passage d'état se fait par le gestionnaire de ressource, ou par le *timer* lié aux tâches périodiques)

➔ L'ordonnanceur utilise les niveaux de priorité natifs (s'il y en a)

- Le nombre de niveaux de priorité natifs est limité selon les OS
- Attention à l'ordre des niveaux
  0 < 1 < 2 ... < 255 (ordre numérique)
  0 > 1 > 2 ... > 255 (ordre numérique inversé)

➔ L'OS peut inclure ou non des primitives de changement de priorité de tâche en cours d'exécution

Ordonnancement classique – FIFO à file unique (sans priorité)

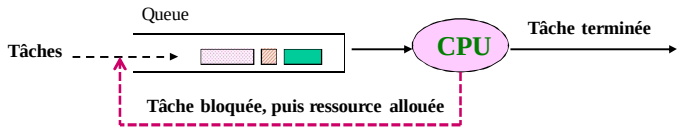
- SCHED_FIFO (*First In First Out*) dite aussi FCFS (*First Come First Served*)

- Les threads à exécuter sont placés dans une file selon leur ordre d'arrivée.

- **Ordo non préemptif** : Le thread en cours d'exécution continue à s'exécuter jusqu'à sa fin ou bien jusqu'à ce qu'il se bloque.

- Lorsque le thread en cours se termine, le thread se trouvant en tête de file d'attente est élu pour s'exécuter.

- **Il est très difficile de garantir des contraintes de temps avec la politique FIFO**

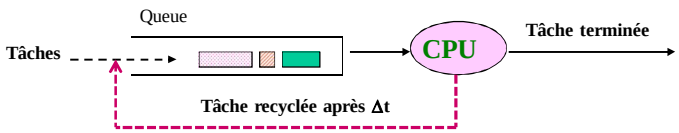


Ordonnancement classique – FIFO à plusieurs files (avec priorité)

- SCHED_FIFO (*First In First Out*) dite aussi FCFS (*First Come First Served*)
  - Il y a une file par priorité.
  - Les threads à exécuter sont placés dans les files selon leur ordre d'arrivée et leur priorité.
  - **Ordo préemptif** : Le thread en cours d'exécution continue à s'exécuter jusqu'à sa fin ou bien jusqu'à ce qu'il se bloque ou jusqu'à ce qu'un thread plus prioritaire demande à s'exécuter.
  - Lorsque le thread en cours se termine, le thread se trouvant en tête de file d'attente la plus prioritaire est élu pour s'exécuter.
  - **Il est difficile de garantir des contraintes de temps avec la politique FIFO**

Ordonnancement classique – Round Robin à file unique (sans priorité)

- SCHED_RR (round robin)
  - RR est une technique d'ordo préemptif de nature.
  - C'est la technique utilisée dans les systèmes à temps partagé (time sharing).
  - Elle s'apparente à FIFO, mais un thread n'a le droit de s'exécuter que pendant un certain quantum **Δt**, ensuite il est préempté et recyclé en queue de la file.
  - **Cycle RR (Δt)** : intervalle de temps tel que chaque thread utilise son quantum.
  - La valeur du cycle RR est un paramètre de configuration de système.
  - **Il est difficile de garantir des contraintes de temps avec la politique SCHED_RR**

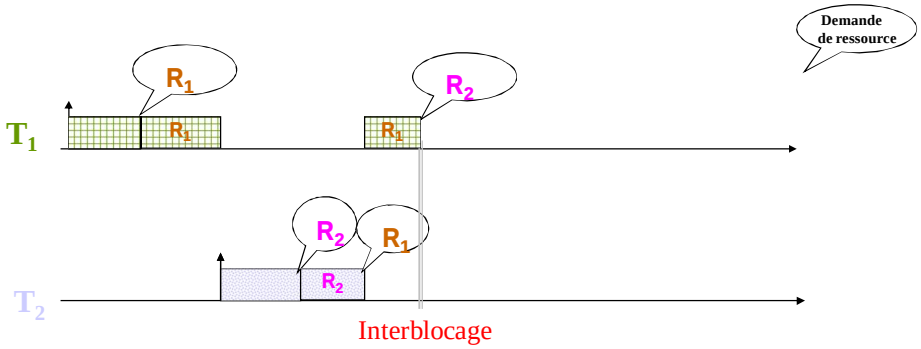


Ordonnancement classique – Round Robin à plusieurs files (avec priorité)

- SCHED_RR (round robin)
  - Il y a une file par niveau de priorité.
  - Le thread en tête de file prioritaire est exécuté jusqu'à sa fin ou jusqu'à son blocage ou jusqu'à épuisement du Δt.
  - Le système sert la file prioritaire jusqu'à ce qu'elle devienne vide avant de passer à la file de priorité inférieure.
- **Cycle RR** : le quantum de temps peut être, ou non, le même pour tous les niveaux de priorité.
- **Il est difficile de garantir des contraintes de temps avec la politique SCHED_RR**

Problèmes inhérents à la programmation concurrente

- Problème de sûreté (*safety*) : Interboclage



Ressources : R_1 et R_2 $Prio_1 < Prio_2$

Problèmes inhérents à la programmation concurrente

- Problème de vivacité (*liveliness*): Famine, non terminaison

  - Suite à une mauvaise conception deux processus n'arrivent jamais à se synchroniser.

    Par exemple, pour réparer un chauffe-eau électrique, le plombier exige que l'électricité soit aux normes (et il faut donc l'intervention de l'électricien). De son côté, l'électricien exige de ne faire les travaux que lorsque les problèmes de fuite d'eau seront résolus (il faut donc l'intervention préalable du plombier).

  - Suite à une mauvaise conception, un groupe de tâches s'accaparent une ressource et empêchent d'autres tâches de les utiliser.

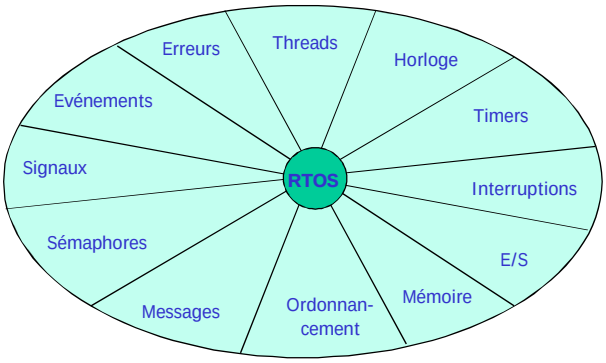
- Problème de reproductibilité des erreurs

Lorsque plusieurs tâches s'exécutent en parallèle, le nombre de comportements possibles devient très élevé. Ainsi si une erreur survient, il est difficile de reproduire cette erreur afin de comprendre son origine.

2. RTOS (concepts de base)

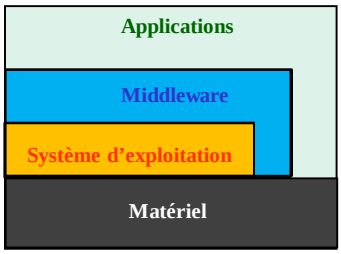
- Exécutif = Noyau de système d'exploitation
- Caractéristiques d'un exécutif temps réel
  - comportement de l'OS prédictible
    - * Ordonnancement de tâches prédictible
    - * Gestion de ressources prédictible
  - dédié à une application spécifique (système embarqué)
  - plus spécialisé qu'un système d'exploitation
  - code de taille plus petite qu'un système classique
  - taille mémoire réduite (services optimisés et ceux non utilisés supprimés)
  - surcoût du système minimal (changement de contexte)
  - nombre élevé d'IT utilisables par les tâches
  - facilité d'insertion et retrait de périphériques

Fonctions de base d'un RTOS



Support d'application temps réel

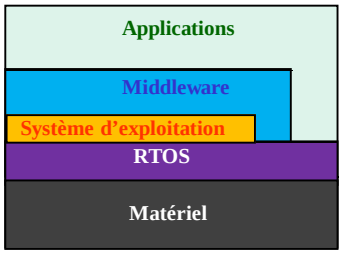
Solution avec OS conventionnel



STR utilisant un OS conventionnel

(approche peu efficace, non recommandée)

Solution avec RTOS



STR fondés sur un RTOS

Classes d'OS pour le temps réel

- OS Conventionnels (Unix, Lunix, Windows) : pas de garantie
- Extensions d'OS conventionnels (RT-Linux)
- RTOS réels
- OS dédiés (écrits pour les besoins de systèmes spécifiques) :
  une tâche qui active les autres, table de scrutation...

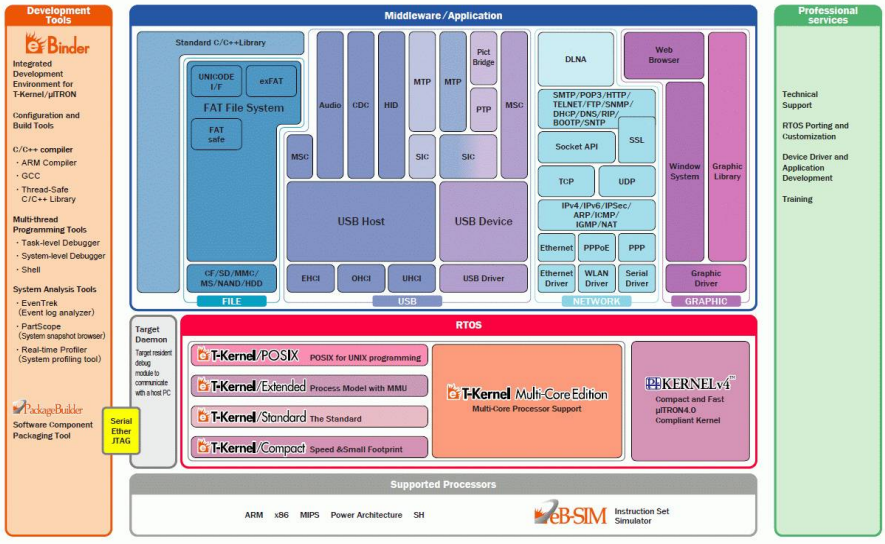
Exemples de Système d'exploitation temps réel

- **Produits commerciaux**
  - Lynx-OS : système Unix à base de thread noyau - compatible avec Linux
    www.lynuxworks.com/
  - QNX : système Unix
    www.qnx.com/
  - Windows CE : système Microsoft temps réel
    www.microsoft.com/technet/prodtechnol/wce/plan/realtime.mspx
  - VxWorks et pSos : Exécutif de Wind river
    www.windriver.com/products/device_technologies/os/vxworks5/
  - Nucleus Real-Time Operating System
    www.mentor.com/proyducts/embedded_software/nucleus_rtos/index.cfm
- **Open sources**
  - RTEMS : www.rtems.org
  - RT-Linux : www.fsmlabs.com/rtlinuxpro.html (extensions de Lunix)
  - KURT : www.ittc.ku.edu/kurt (extensions de Lunix)
  - RTAI : www.aero.polimi.it/rtai/news/index.html (extensions de Lunix)

3. Standard POSIX pour le temps réel

- POSIX = Portable Operating System Interface for uniX
- **Objectifs :** Définir une interface standard entre les applications et l'OS pour rendre les applications (TRE) portables
- Services proposés
  - Threads, ordonnancement, synchronisation
  - Gestion du temps
  - Files de messages, signaux
  - E/S asynchrones
  - Gestion de la mémoire
- **POSIX 1003.1 : interface de base d'accès à l'OS.**
- **POSIX 1003.1b** (appelé aussi 1003.4) – 1993 **: Extensions temps réel** (ordonnancement à priorité, signaux TR, horloges et timers, sémaphores, messages, E/S asynchrones…)
- **POSIX 1003.1c** (appelé aussi 1003.4) – 1993 **: Extensions de threads** (création/destruction de threads, synchronisation et ordonnancement de threads…)
- **Beaucoup d'implantations de POSIX sont disponibles : Lynx, VxWorks, RTEMS…**

4. Exemple de plateforme : eSOL.com



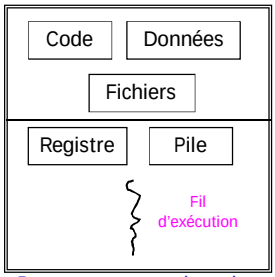
Chapitre 3

Fondements pour la programmation des systèmes temps réel et embarqués

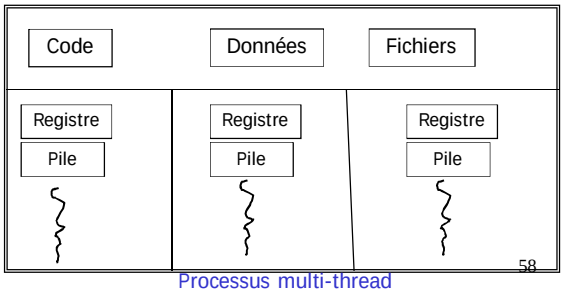
Eléments de la norme POSIX

1. Programmation concurrente en C avec la norme Posix
(création et terminaison de thread)

- La norme POSIX 1003.c (1995) définit une API pour manipuler des tâches appelées *threads* ('pthreads') avec le langage C.
- Les définitions liées aux threads se trouvent dans le fichier `<pthread.h>`
- Chaque thread a ses registres et sa pile d'exécution. Mais tous les threads d'un même processus partagent le même espace virtuel, les mêmes actions événementielles, les mêmes descripteurs de fichiers et d'autres ressources du processus.



Processus mono-thread



Processus multi-thread

- Un thread (tâche) est défini, du point de vue programmeur, par son identifiant système de type `pthread_t`
- Une tâche POSIX n'est pas définie mais créée par un appel de la primitive `pthread_create`.

L'appel de `pthread_create` crée une nouvelle tâche qui s'exécute en parallèle avec celle qui l'a créée.

```
int pthread_create ( pthread_t *thread, const pthread_attr_t *attr,
                     void *(*start_fnc)(void *), void *arg)
```

En cas de succès, la fonction renvoie la valeur `0`. La nouvelle tâche est lancée automatiquement après sa création. En cas d'échec de création, la fonction renvoie un code d'erreur.

- L'argument `attr` indique les attributs de la tâche. Il peut être égal à `NULL`.
- La fonction `start_fnc` est la première fonction exécutée par la tâche quand celle-ci est démarrée.
- `arg` contient les paramètres passés à une tâche au moment de sa création.

- Le type **`pthread_attr_t`** est défini par les attributs de thread suivants :

```
typedef struct
{  int __detachstate;      // indique si ce thread peut se synchroniser (par join)
                           // avec un autre

   int __schedpolicy;      // Politique d'ordonnancement du thread (FIFO, RR, Other)

   struct __sched_param __schedparam;       // Paramètres d'ordo du thread
                  // en général, il y a un seul paramètre : la priorité

   int __inheritsched;     // Attribut d'héritage de la politique d'ordo du père
   int __scope;            // indique la portée de la priorité du thread (en général,
                           // la priorité est globale au système)

   size_t __guardsize;     // Infos sur la pile d'exécution

   int __stackaddr_set;    //  "     "      "     "

   void *__stackaddr;      //  Adresse en mémoire de la pile d'exécution du thread

   size_t _stacksize;      //  Taille de la pile d'exécution du thread

} pthread_attr_t;
```

Terminaison de tâche

- Une tâche POSIX se termine soit en appelant explicitement la fonction **`pthread_exit`** ou implicitement lorsque la fonction `start-routine` s'achève
- Si le programme principal termine normalement, les tâches Posix créées par celui-ci sont terminées par un appel implicite à la fonction système **`exit`**.
- Si l'on veut terminer proprement un programme créant des tâches Posix, il est nécessaire d'attendre la terminaison de celles-ci en utilisant la primitive **`pthread_join`**

```
int pthread_join(pthread_t th_a_attendre, void **tache_return);
```


Autres primitives de manipulation de threads

- Initialisation d'attributs de thread : `pthread_attr_init`
- Destruction d'attributs de thread : `pthread_attr_destroy`
- Obtention de l'adresse de pile de thread : `pthread_getstackaddr`
- Modification de l'adresse de pile de thread : `pthread_setstackaddr`
- Obtention de la taille de pile de thread : `pthread_getstacksize`
- Modification de la taille de pile de thread : `pthread_setstacksize`
- Test d'égalité de deux identifieurs de threads : `pthread_equal`
- Terminaison d'un thread : `pthread_exit`
- Obtention du Pid de l'appelant : `pthread_self`

Exemple de programme multi-tâches

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *affichCar(void *arg){
        char c;
        int k;
        c = * (char *)arg;
        for (k=1; k<50; k++){
                putchar( c);
                }
        }
```


// Création de deux tâches identiques

```
int main(void)
   {    char *leCar; int i;
        pthread_t tache_Posix_B;
        pthread_t tache_Posix_C;
        leCar = (char*) malloc(1*sizeof(char)); *leCar = 'B';
        pthread_create( &tache_Posix_B, NULL, affCar, (void*) leCar);
        leCar = (char*) malloc(1*sizeof(char)); *leCar = 'C';
        pthread_create( &tache_Posix_C, NULL, affCar, (void*) leCar);
        for (i=0; i<100; i++){
                putchar('Z');
                }
        pthread_join (tache_Posix_B, NULL);
        pthread_join (tache_Posix_C, NULL);
  }
```

Si on ne met pas ces deux instructions, les deux tâches B et C seront terminées à peine créées (elles n'auront pas le temps de s'exécuter)

2. Synchronisation et communication inter-tâches

- Différents mécanismes sont utilisables pour la synchronisation
  - **Interruptions** (niveau hardware)
  - **Signaux** (ex. signaux Unix)
  - **Sémaphores** (mal utilisés les sémaphores conduisent à des interblocages...)
  - **Moniteurs** (mécanisme de haut niveau; évite certaines erreurs de conception)
  - **Autres :** verrous...

- Rappels sur les sémaphores (Dijkstra 1965)
  - Un compteur entier (S_CPT)
  - Une file d'attente S_Attente
  - Deux opérations P et V et éventuellement d'une fonction d'initialisation S_init

```
P(S) : // prend le sémaphore; bloquant si sémaphore déjà pris
       S_CPT --;
       Si (S_CPT < 0) Alors
          Bloquer l'appelant et le mettre dans la file S_Attente;
       fin si

V(S) : // rend le sémaphore; jamais bloquant
         S_CPT ++;
         Si (S_CPT <= 0) Alors
            Choisir un processus dans la file S_Attente,
            le retirer de celle-ci et le débloquer;
        fin si

S_init(S, Val) : // initialise le compteur interne à la valeur Val
         S_CPT <-- Val;
```

Mutex de Posix pour l'exclusion mutuelle

- Concepts et principes de base

  - Un *mutex* est un sémaphore binaire (initialisé par défaut à 1) pouvant prendre un des deux états : "lock" (verrouillé) ou "unlock" (dévérrouillé)

  - Un *mutex* est utilisé pour protéger l'accès à une ressource/section critique

  - Un *mutex* ne peut être partagé que par des threads d'un même processus

  - Un *mutex* ne peut être verrouillé que par un seul thread à la fois.

  - Un thread qui tente de verrouiller un *mutex* déjà verrouillé est suspendu jusqu'à ce que ce *mutex* soit déverrouillé.

  - Un *mutex* est une variable de type `thread_mutex_t`

67

- Déclaration et initialisation d'un mutex

  - Il existe une constante `PTHREAD_MUTEX_INITIALIZER` de type `thread_mutex_t` permettant une déclaration avec initialisation statique du *mutex*

```
pthread_mutex_t monMutex = PTHREAD_MUTEX_INITIALIZER;
```

  - Un *mutex* peut également être initialisé par un appel de la primitive

```
int pthread_mutex_init(pthread_mutex_t *mutex,
        const pthread_mutexattr_t *mutexattr);
typedef struct { int __mutexkind; } pthread_mutexattr_t;
```

    Avec une initialisation par défaut, `mutexattr` vaut NULL

    ex :`pthread_mutex_init(&monMutex, NULL);`

  - Un *mutex* non utilisé peut (doit) être supprimé par `pthread_mutex_destroy`

```
int pthread_mutex_destroy (pthread_mutex_t *mutex)
```

68

Verrouillage d'un mutex

- Un *Mutex* peut être verrouillé par la primitive `pthread_mutex_lock`

```
init pthread_mutex_lock(pthread_mutex_t *mutex);
```

* Si le *mutex* est déverrouillé, il devient verrouillé.
  Si le *mutex* est verrouillé, l'appelant est bloqué jusqu'à ce que *mutex* soit verrouillé.

- Une autre alternative au `pthread_mutex_lock` bloquant est le `pthread_mutex_trylock` qui est non bloquant (qui vérrouille le *mutex* s'il est déverrouillé et qui renvoie une erreur si le *mutex* est déjà verrouillé)

```
init pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Posix 1003.b de 1999 définit une version d'attente de mutex temporisée (ce qui limite le blocage infini) :

```
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
    const struct timespec *abs_timeout)
```

69

Déverrouillage d'un mutex

- Un *Mutex* peut être déverrouillé par la primitive `thread_mutex_lunock`

```
init pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  * Si le *mutex* est déjà déverrouillé, il reste déverrouillé.
  * Si le *mutex* est verrouillé, un des threads en attente de ce *mutex* est débloqué.

- L'opération de déverrouillage est toujours non bloquante pour l'appelant.

Autres opérations sur un mutex

- `pthread_mutexattr_destroy` : détruit des attributs de *mutex* :
- `pthread_mutexattr_getshared` : détermine si un mutex est partageable par des processus
- `pthread_mutexattr_setshared` : modifie l'attribut de partage d'un mutex
- `pthread_mutexattr_init` : initialise les attributs d'un mutex

70

Exemple d'utilisation de *mutex*

```
.../..
pthread_mutex_t monMutex = PTHREAD_MUTEX_INITIALIZER;
void LitRegistre(ValeurRegistre R){
   int i; for (i=0; i<TAILLE_REGISTRE; i++) R[i] = LeRegsitre[i];
}
void EcritRegistre(ValeurRegistre R){
   int i; for (i=0; i<TAILLE_REGISTRE; i++) LeRegsitre[i] = R[i];
}
void f(void){
   int i, j; ValeurRegistre RLocal;
   for(j=0; j < 1000000; j++){
       pthread_mutex_lock( &monMutex );
       LitRegistre( RLocal );
       for(i=0; i<TAILLE_REGISTRE; i++){ RLocal[i] ++; }
       EcritRegistre( RLocal );
       pthread_mutex_unlock( &monMutex );
   }
}
```

Section critique

Sémaphores de Posix

- Notion de sémaphore Posix
  - Un sémaphore Posix est un sémaphore à compte pouvant être partagé par plusieurs threads de plusieurs processus (selon les implémentations)
  - Le compteur associé à un sémaphore peut donc prendre des valeurs plus grandes que 1 (contrairement à un mutex)
  - La prise d'un sémaphore dont le compteur est négatif ou nul bloque l'appelant
  - Les définitions nécessaires à la manipulation de sémaphore se trouvent dans le fichier entête `<semaphore.h>`
- Déclaration et initialisation d'un sémaphore
  - Un sémaphore est une variable de type **`sem_t`**
  - Un sémaphore est initialisé par la primitive :

```
int sem_init(sem_t *sem, int pshared, unsigned int valeur);
```

  * si "`pshared`" vaut 0 le sémaphore ne peut pas être partagé qu'entre tâches d'un même processus
  * si "`pshared`" n'est pas égal à 0, le sémaphore peut être partagé entre plusieurs processus
  * `valeur` définit la valeur initiale de ce sémaphore (positif ou nul)

- Prise de contrôle et libération de sémaphore Posix
  - Les deux opérations P et V sont implantées par :

```
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

  - Il existe également une version non bloquante de la primitive P :

```
int sem_trywait(sem_t *sem);
```

    qui retourne 0 si quand la prise est possible (et non bloquante) et qui retourne EAGAIN sinon (dans le cas où l'appel normal serait bloquant)

  - Posix 1003.b de 1999 définit une version de P temporisée (ce qui limite le blocage infini) :

```
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

- Autres opérations sur les sémaphores
  - destruction de sémaphore : `sem_destroy`
  - renvoi de la valeur courante du compteur d'un sémaphore : `sem_getvalue`
  - établissement d'un lien entre un sémaphore nommé et un thread : `sem_open`
  - rupture du lien entre un sémaphore et un thread : `sem_close`

Variables conditionnelles de Posix

- Objectifs
  - Les variables conditionnelles servent à mettre un thread en attente de la satisfaction d'une condition (dépassement d'un seuil de température par exemple) qui dépend de l'état d'une variable partagée.
  - Les variables conditionnelles sont utilisées conjointement avec les *mutex*. Chaque variable conditionnelle est liée à un *mutex* qui la protège.
  - Avant de tester une variable conditionnelle, le thread doit verrouiller l'accès à la variable par le *mutex* associé à cette variable.
  - Quand un thread se bloque, par `pthread_cond_wait`, en attente de la satisfaction d'une condition, le *mutex* associé à la variable conditionnelle est automatiquement libéré par le système pour permettre à un autre thread de rendre la condition vraie.
  - Quand un thread est réveillé, suite à un `pthread_cond_signal` ou `pthread_cond_broadcast`, le *mutex* est toujours en position verrouillé (le thread réveillé doit donc le déverrouiller).
  - Les variables conditionnelles sont utilisées pour la **signalisation** pas pour l'exclusion mutuelle.

- Déclaration, initialisation et manipulation de variable conditionnelle

- Type de variable conditionnelle : **pthread_cond_t**

- On peut définir une variable conditionnelle de manière statique avec des attributs par défaut :

```
pthread_con_t NomCond = PTHREAD_COND_INITIALIZER;
```

- Une variable conditionnelle peut aussi être initialisée de manière dynamique par

```
int pthread_cond_init(pthread_cond_t *cond,
        const pthread_condattr_t *attr = NULL)
struct pthread_condattr_t { int __dummy; } pthread_condattr_t;
```

- Une variable conditionnelle est détruite par

```
int pthread_cond_destroy (pthread_cond_t *cond)
```

- Un thread se bloque en attente qu'une variable conditionnelle devienne vraie

```
int pthread_cond_wait pthread_cond_t *cond, pthread_mutex_t *mutex)
```

- Un thread peut signaler une variable conditionnelle par

```
int pthread_cond_signal (pthread_cond_t *cond)
```

(si aucun thread est en attente de la variable, le signal est perdu contrairement à V(sem))

75

- Un thread peut signaler une variable conditionnelle à tous les threads en attente par

```
int pthread_cond_braodcast (pthread_cond_t *cond)
```

- Posix 1000.3 de 1999 définit aussi l'attente temporisée sur une condition pour éviter le blocage infini

```
int pthread_cond_timedwait(pthread_cond_t *cond,
        pthread_mutex_t *mutex, const struct timespec *abstime)
```

- Autres primitives relatives aux variables conditionnelles

- `pthread_cond_attr_init` : initialise les attributs d'une var-cond
- `pthread_condattr_destroy` : détruit des attributs de variable conditionnelle
- `pthread_condattr_getshared` : test si une var-cond est partageable entre processus
- `pthread_condattr_setshared` : modifie l'attribut de partage d'une var-cond

76

Exemple 1 d'application avec variable conditionnelle

- Dans l'exemple suivant un thread incrémente un compteur, les autres attendent qu'il franchisse le seuil de 100.

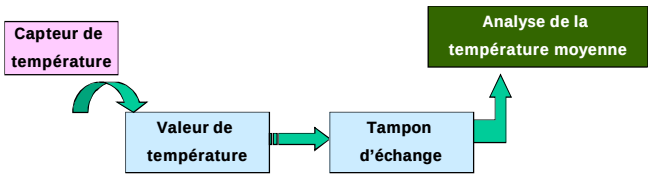
```
        // Thread de calcul
•••
pthread_cond_t VarCond =
PTHREAD_COND_INITIALIZER ;
•••
while (...) {
   •••
  pthread_mutex_lock(&Verrou);
  Compteur++;
  if (Compteur > 100)
   pthread_cond_broadcast (&VarCond);
   pthread_mutex_unlock (&Verrou);
   •••
}
•••
```

```
// Threads qui attendent le //
franchissement du seuil de 100
•••
pthread_mutex_init(&Verrou, NULL);
pthread_cond_init(&VarCond, NULL);
•••
pthread_mutex_lock (&Verrou);
while (100 < Compteur)
 pthread_cond_wait(&VarCond, &Verrou);
printf ("Seuil atteint! "\n);
pthread_mutex_unlock (&Verrou);
```

77

Exemple 2 d'application avec variable conditionnelle

- **Description de l'exemple**
  - Un robot doit envoyer régulièrement des informations de température
  - Il dispose pour cela d'un capteur de température
  - On décide d'attribuer à une tâche le travail de lecture périodique du capteur (toutes les secondes) et de transmettre à une autre tâche la valeur moyenne sur 10 mesures
  - La seconde tâche fait un pré-traitement de cette moyenne et envoie à son rythme ces informations vers un central
  - On modélise dans ce problème le capteur et les deux tâches; l'interaction du robot et du superviseur central n'est pas abordée.



78

- **Solution avec mutex et variable conditionnelle**

```
/* ____________________________________________________________
Exemple d'application avec variable conditionnelle POSIX.
Description de l'exemple :
- L'application est composée de deux tâches : tacheLecture et tacheCalcul.
- tacheLecture lit périodiquement la mesure de température et transmet à
  une autre tâche la valeur moyenne sur 10 mesures.
- tacheCalcul fait un pré-traitement de la moyenne reçue et envoie à
  son rythme son résultat vers un central non programmé ici.

Compiler avec : gcc -pthread -lrt .....;
____________________________________________________________*/

#include <stdio.h>
#include <pthread.h>
#include <time.h>
#include <stdlib.h>
typedef unsigned char int;

unsigned int periodeLecture = 1; // lire la températures toutes les 1 sec.
unsigned int nombreDonnees = 10; // 10 lectures avant de transmettre
int valeurTampon = 0;
int valeurPrete = 0;
```

79


```
pthread_mutex_t mutexTampon = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t attenteDonnee; /* variable conditionnelle, pour la gestion du
          tampon d'échange, initialisée dans le programme principal avant de
          lancer les tâches. */

// Fonction de simulation du capteur de température
int CapterTemperature()
 {       int  temperature;
         temperature = rand();
          return temperature;
 }

// Code associé à la gestion du tampon : utilisation d'une variable
conditionnelle
void Tampon_metAJour(int v)
 {
         pthread_mutex_lock(&mutexTampon);
         valeurTampon = v;
         valeurPrete = 1;
         pthread_cond_signal(&attenteDonnee);
         pthread_mutex_unlock(&mutexTampon);
 }
```

80

```
void Tampon_lectureQuandPret(int *v)
 {
        pthread_mutex_lock( &mutexTampon);
        while ( !valeurPrete)
        {  pthread_cond_wait(&attenteDonnee, &mutexTampon); }
        *v = valeurTampon;
        valeurPrete = 0;
        pthread_mutex_unlock( &mutexTampon);
}

/* Code associé à la tâche de lecture régulière du registre
   (c'est une version simplifiée pour la gestion du temps) */
void codeLectureReguliere(void)
 {      int  somme; int CPT = 0;
        printf("Entrée dans le thread tacheLecture \n");
        while(1)
        { sleep(periodeLecture);
          somme += CapterTemperature;
          CPT ++;
          if (CPT == nombreDonnees)
                { somme /= nombreDonnees;  Tampon_metAJour(somme);
                  CPT = somme = 0; }
        }
 }
```

81


```
// Code associé à la tâche de calcul (analyse de la tempéature)
void codeCalcul(void)
        { int moyenneTemp;   int leMin = 255;     int leMax = 0;
          printf("Entrée dans le thread tacheCalcul \n");
          while(1)
                { Tampon_lectureQuandPret(&moyenneTemp);
                  if (moyenneTemp < leMin) {leMin = moyenneTemp;}
                  if (moyenneTemp > leMax) {leMax = moyenneTemp;}
                  printf("Valeur température = %d \n", moyenneTemp);
                  printf("min = %d, max = %d \n", leMin, leMax);
                }
        }
 int main (int argc, char *argv[])
        { pthread_t tacheCalcul;
          pthread_t tacheLecture;
          pthread_cond_init( &attenteDonnee, NULL);
          pthread_create(&tacheLecture, NULL, (void *(*)()) codeLectureReguliere,
                                        NULL);
          pthread_create(&tacheCalcul, NULL, (void *(*)()) codeCalcul, NULL);
          pthread_join(tacheLecture, NULL); // ne doit jamais arriver
          pthread_join(tacheCalcul, NULL); // ne doit jamais arriver
          printf("Bizarre : terminaison inattendue des deux tâches !!!! \n");
          return -1;
        }
```

82

Signaux de Posix

- Principes
  - Les différentes occurrences d'un même signal restent pendantes (le nombre de signaux reçus est égal au nombre de signaux émis).
  - La priorité liée au signal est respectée, avec celle du thread, dans la gestion de la file d'attente des signaux : les threads prioritaires son réveillés selon leur priorité
  - Les nouveaux signaux (par rapport à Unix) sont au nombre de `RTSIG_MAX` et sont numérotés de `SIGRTMIN` à `SIGRTMAX`.
  - Les définitions nécessaires à l'utilisation des signaux se trouvent dans le fichiers `<signal.h>`

- Remarques

1) Attention l'utilisation des signaux avec les threads peut conduire à des erreurs difficiles à détecter au moment de l'écriture du code. Ceci est dû au fait que les threads d'un process partagent les mêmes infos sur les signaux.

2) L'utilisation des signaux doit être conforme à l'utilisation traditionnelle des signaux sous UNIX, En particulier pour la définition des actions événementielles

83

- **Liste des signaux POSIX**

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGABRT	7) SIGEMT	8) SIGFPE
9) SIGKILL	10) SIGBUS	11) SIGSEGV	12) SIGSYS
13) SIGPIPE	14) SIGALRM	15) SIGTERM	16) SIGUSR1
17) SIGUSR2	18) SIGCHLD	19) SIGPWR	20) SIGWINCH
21) SIGURG	22) SIGIO	23) SIGSTOP	24) SIGTSTP
25) SIGCONT	26) SIGTTIN	27) SIGTTOU	28) SIGVTALRM
29) SIGPROF	300) SIGXCPU	31) SIGXFSZ	

33) SIGRTMIN	34) SIGRTMIN+1	35) SIGRTMIN+2	36) SIGRTMIN+3 37)
SIGRTMIN+4	38) SIGRTMIN+5	39) SIGRTMIN+6	40) SIGRTMIN+7 41)
SIGRTMIN+8	42) SIGRTMIN+9	43) SIGRTMIN+10	44) SIGRTMIN+11 45)
SIGRTMIN+12	46) SIGRTMIN+13	47) SIGRTMIN+14	48) SIGRTMIN+15 49)
SIGRTMAX-15	50) SIGRTMAX-14	51) SIGRTMAX-13	52) SIGRTMAX-12 53)
SIGRTMAX-11	54) SIGRTMAX-10	55) SIGRTMAX-9	56) SIGRTMAX-8 57)
SIGRTMAX-7	58) SIGRTMAX-6	59) SIGRTMAX-5	60) SIGRTMAX-4 61)
SIGRTMAX-3	62) SIGRTMAX-2	63) SIGRTMAX-1	64) SIGRTMAX

Signaux "Non Posix" : SIGEMT SIGIO SIGPWR IGWINCH

84

- Primitives des manipulation de signaux

- L'attente de signal se fait par de trois manières :

1) Attente simple d'un signal

```
int pthread_sigwait (const sigset_t *set, int *sig)
```

2) Attente de signal accompagnée d'information

```
int pthread_sigwaitinfo (const sigset_t *set, siginfo_t *info)
```

3) Attente de signal temporisée :

```
int pthread_sigtimedwait (const sigset_t *set, siginfo_t *info,
    const struct timespec *timeout)
```

- Masquage de signaux

```
int pthread_sigmask (int how, const sigset_t *newmask, sigset_t *oldmask)
```

- Emission de signal à un thread explicitement désigné

```
int pthread_kill (pthread_t thread, int signumber)
```


Exemple avec signaux et sémaphores

- Dans l'exemple du programme suivant, on utilise un sémaphore pour réveiller 5 threads dans une fonction de capture de signal (de timer).

- On notera que le sémaphore est initialisé à 0 (par **sem_init**) pour bloquer tout processus qui exécute un **sem_wait**.

- Le programme principal arme un timer cyclique.

- Chaque occurrence du signal du timer permet de débloquer un processus (par **sem_post**).

- Le programme se termine quand chaque thread a été débloqué (réveillé) 5 fois.

```
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <fcntl.h>
#include <semaphore.h>
#include <signal.h>
#include <time.h>
#include <sys/errno.h>
```

```
sem_t semafor;
```

```
sem_t semafor;

 // Fonction de capture de signal
        void signal_catcher (int sig)
          { printf(">>>> Fonction Catcher exécutée \n");
         if (sem_post(&semafor)==-1) printf("Erreur sur : post sempahore \n");}
```

```
// Fonction Start de thread; cette routine attend le sémaphore.
        void *sem_waiter(void *arg)
        { int number = (int)arg; int CPT;
        // Chaque thread attend 5 fois
             for (CPT=1; CPT <=5; CPT++)
                 { while (sem_wait(&semafor)==-1)
                   if (errno!= EINTR) printf("Erreur sur : wait on sempahore \n");
                   printf("Thread #%d réveillé pour la (%d)ème fois...\n", number, CPT);
                 }
           return NULL ; }
```



```
int main (int argc, char *argv[])

{       int thread_CPT, status;
        struct sigevent sig_event;
        struct sigaction sig_action;
        sigset_t sig_mask;
        timer_t my_timer;
       struct itimerspec ti;

        pthread_t sem_waiters[5];

        // sem_init(&sempahore, 0, 0);

 // Création de 5 threads pour attendre le sémaphore
        for(thread_CPT = 0; thread_CPT < 5; thread_CPT++)
        {       printf("Création thread #%d \n", thread_CPT);
                        status = pthread_create (
                                &sem_waiters[thread_CPT], NULL,
                                sem_waiter, (void*)thread_CPT);
                        if (status !=0)
                                printf("Erreur de création de thread \n");
        }
```

```
// Armer un timer, en utilisant le signal SIGRTMIN, qui doit se déclencher
// toutes les 2 secondes

       sig_event.sigev_value.sival_int = 0;
        sig_event.sigev_signo = SIGRTMIN ;
        sig_event.sigev_notify = SIGEV_SIGNAL ;

        if (timer_create(CLOCK_REALTIME, &sig_event, &my_timer)==-1)
                printf("Erreur de création de timer \n");

        sigemptyset (&sig_mask);
        sigaddset (&sig_mask, SIGRTMIN);
        sig_action.sa_handler = signal_catcher;
        sig_action.sa_flags = 0;
        if (sigaction(SIGRTMIN, &sig_action, NULL) == -1)
                printf("erreur d'initialisation d'action de signal \n");

       ti.it_interval.tv_sec = 2;
        ti.it_interval.tv_nsec = 0;
        ti.it_value.tv_sec = 2;
        ti.it_value.tv_nsec = 0;

        if (timer_settime(my_timer, 0, &ti, NULL) == -1)
                printf("Erreur d'armement du timer \n");
```

89


```
// Attente de la terminaison de tous les threads.
        for (thread_CPT = 0; thread_CPT < 5; thread_CPT++)
        {       status = pthread_join(sem_waiters[thread_CPT], NULL);
                if (status != 0) printf("Erreur de JOIN de thread.\n");
         }

        return 0;
 }
```

90

Interface pour la gestion du temps

`clockid-t` : type pour manipuler l'horloge

`timer_t` : type pour manipuler les timers

- Primitives de l'API de gestion du temps

- `int clock_settime (clockid_t clock_id, const struct timespec *tp)` : initialise l'horloge
- `int clock_gettime (clockid_t clock_id, struct timespec *tp)` : lit la valeur de l'horloge
- `int clock_getres (clockid_t clock_id, struct timespec *res)` : renvoie la résolution de l'horloge
- `int timer_create (clockid_t clock_id, struct sigevent *evp, timer_t *timerid)` : crée un timer en ayant comme base l'horloge donnée en argument
- `int timer_delete ( timer-t timerid )` : détruit un timer
- `int timer_settime( timer_t timerid, int flags, const struct itimerspec *value, struct itimerspec *ovalue )` : arme un timer avec une valeur de temporisation
- `int timer_gettime ( timer_t timerid, struct itimerspec *value )` : lit le temps restant à un timer avant d'expirer

- Primitives de l'API (suite)

- `int timer_getoverrun ( timer_t timerid )` : lit le temps de dépassement d'un timer
- `int nanosleep( const struct timespec *rqtp, struct timespec *rmtp )` :

  suspension du thread appelant pendant une durée spécifiée par `rqtp`.

  `rmtp` est généralement égal à NULL.

- Posix 1003.b de 1999 définit la fonction suivante qui permet d'obtenir l'Id d'une horloge :

```
int clock_getcpuclockid (pid_t pid, clockid_t *clock_id);
```

- Utilisation de l'API

- L'utilisation des primitives relatives au temps nécessite le fichier entête `<time.h>`

- La constante `_POSIX_CLOCKRES_MIN`, définie dans `<time.h>`, spécifie la résolution (granularité) de l'horloge temps réel. Sa valeur dépend de la cible (par défaut c'est 20 ms).

Interface pour la gestion de messages Posix

- Principes de la communication par messages

Dépôt et retrait de messages à partir d'une file (avec éventuellement un ordre –position- des messages dans la file que l'on spécifie dans les opérations *mq_send* et *mq_receive*).

- Primitives de l'API

- `mqd-t mq_open ( const char *name, int oflag, ... )` : crée une file de messages et la lie à un processus. `oflag` spécifie les droits sur la file (écriture, lecture ou les deux).

- `int mq_close ( mqd_t mqdes )` : ferme une file de messages

- `int mq_setattr ( mqd_t mqdes, const struct mq_attr *mqstat, struct mq-attr *omqstat )` : change les caractéristiques d'une file de messages (taille maxi de messages, nombre maxi de messages...)

- `int mq_getattr ( mqd-t mqdes, struct mq-attr *mqstat )` : renvoie les attributs d'une file de messages

- `int mq_unlink ( const char *name )` : supprime une file de messages

93

- Primitives de l'API (suite)

- `int mq_send ( mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio )` : rajoute un message dans une file

- `int mq_receive ( mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio )` : extrait un message d'une file

- `int mq_notify( mqd_t mqdes, const struct sigevent *notification )` : avertit un thread de l'arrivée d'un message dans une file.

- Posix 1003.b de 1999 définit des primitives de messages temporisées
  - * **`mq_timedsend`** : en cas de manque de place dans la file, une quantité de temps est spécifiée pour indiquer l'attente maxi avant de déposer un message
  - * **`mq_timedreceive`** : permet de recevoir (au bout d'un délai fixé) un message, s'il n'y a pas de message, retour à l'appelant au bout du temps spécifié.

94

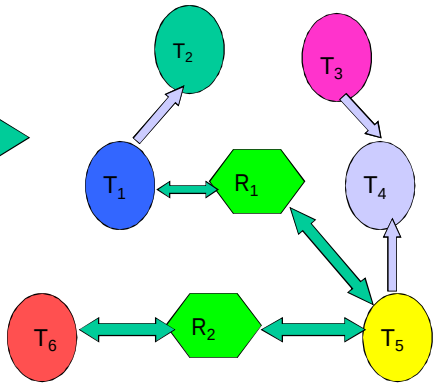
Chapitre 4

Ordonnancement de tâches temps réel

Ordonnancement et standard Posix

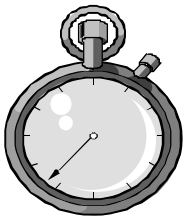
1. Caractéristiques des tâches temps réel

Document de conception

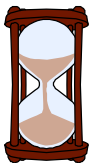


Ensemble de tâches

Classes de contraintes



Top départ



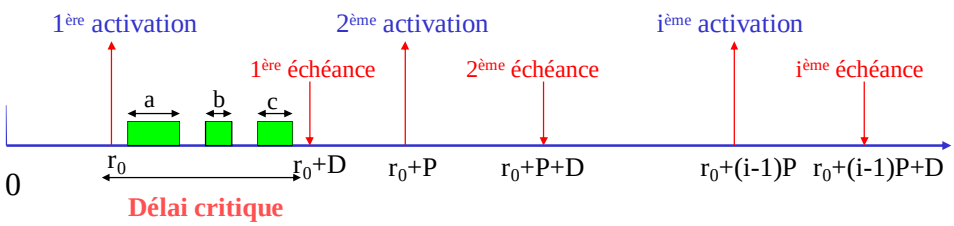
Durée d'exécution

Echéance

➔ **Contraintes individuelles**

➔ **Contraintes collectives**

Modèle simple (tâches périodiques)



r_0 : date de première activation

C : pire durée d'exécution = Max(a) + Max(b) + Max(c)

D : délai critique

P : période **Si P = D : échéance sur requête**

1. Contraintes des tâches temps réel

- **Charge processeur d'une application TR**

$u_i = \frac{C_i}{P_i}$: **Pourcentage de l'activité du processeur dédiée à la tâche T_i**

$U = \sum_{i=1}^{n} \frac{C_i}{P_i}$: **Charge processeur = taux d'utilisation du processeur pour les tâches périodiques**

U > m ⇒ application non ordonnançable sur m processeurs
(U ≤ m : Condition nécessaire, mais pas suffisante)

1. Contraintes des tâches temps réel

- **Type de tâche (critique ou non)**
- **Priorité de tâche**
- **Préemptible ou non**
- **Contraintes temporelles**
  - **Tâche périodique ou apériodique**
  - **Période**
  - **Instant d'arrivée (fixe, cyclique, aléatoire…)**
  - **Durée d'exécution (Worst Case Execution Time …)**
  - **Echéance absolue (date au plus tard de terminaison)**
  - **Echéance relative (temps de réponse maximum)**
  - **Autres**
- **Autres contraintes**

Contraintes collectives

- **Contraintes de relations entre tâches**
  - **Précédence**
  - **Partage de ressources**
- **Contraintes de répartition**
- **Contraintes de tolérance aux fautes**

2. Algorithmes d'ordonnancement temps réel

Notion d'ordonnancement TR

- **Ordonnancement = liste (partiellement) ordonnée d'accès aux ressources partagées**
- **Entités ordonnançables : tâches, threads, processus, messages, ...**
- **Ordonnancement valide : Ordonnancement qui satisfait les contraintes temporelles**
- **L'ordonnancement = discipline qui puise ses racines dans la recherche opérationnelle**
- **Priorités importantes souhaitées (ou requises)**
  - **Optimalité** (capacité à trouver une solution faisable si elle existe)
  - **Stabilité** (capacité à ne pas remettre en cause l'ordonnançabilité après relaxation des contraintes des tâches)
  - **Faible complexité** (un coût d'exécution de l'algorithme d'ordo faible/négligeable)

- **Il existe de TRES nombreux algorithmes d'ordonnancement :**
  statique/dynamique, hors/en ligne, préemptif/non préemptif, à priorités statiques/dynamiques, local/global, avec/sans heuristiques ...

- **Algorithmes à étudier dans ce cours**

  Rate Monotonic (RM) : fondé sur les périodes

  Earliest Deadline First (EDF) : fondé sur les échéances

  Sporadic Server : pour les tâches apériodiques avec échéance

  Priority Inheritance Protocol : pour éviter l'inversion de priorité

  Priority ceiling Protocol : pour éviter les interblocages

Rate Monotonic

Principe
Priorité = F(Période)
Plus petite période ⇒ Plus prioritaire

La priorité est inversement proportionnelle à la période

RM est optimal (pour $P_i = D_i$)

Condition d'ordonnançabilité (pour $P_i = D_i$) :

- **CS (théorème de la limite d'utilisation - Lui 73)**

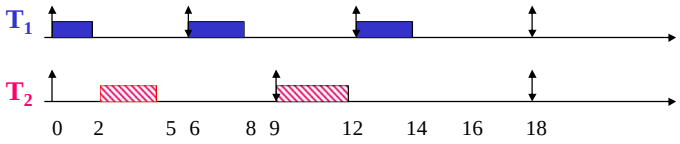
$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}} - 1)$$

RM : exemple 1

$r_1 = r_2 = 0$ $\quad D_1 = P_1 = 6 \quad D_2 = P_2 = 9$

$C_1 = 2 \quad C_2 = 3$

$Prio_2$ < $Prio_1$



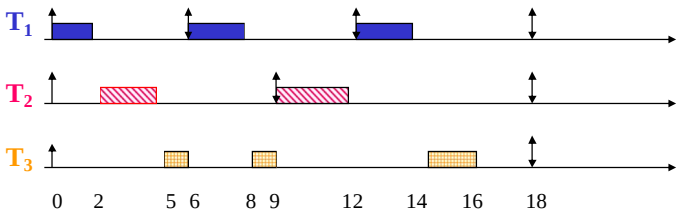
$U = 0,67 < 2(2^{1/2} - 1) = 0,83 \Rightarrow$ CS vérifiée

RM : exemple 2

$r_1 = r_2 = r_3 = 0$ $\quad D_1 = P_1 = 6 \quad D_2 = P_2 = 9 \quad D_3 = P_3 = 18$

$C_1 = 2 \quad C_2 = 3 \quad C_3 = 4$

$Prio_3$ < $Prio_2$ < $Prio_1$



$U = 0,89 > 3(2^{1/3} - 1) = 0,78 \Rightarrow$ CS non vérifiée

Earliest Deadline First

Principe
Echéance la plus proche ⇒ Plus prioritaire

La priorité est inversement proportionnelle à l 'échéance

EDF est optimal pour les systèmes de tâches indépendantes (pour $P_i = D_i$)

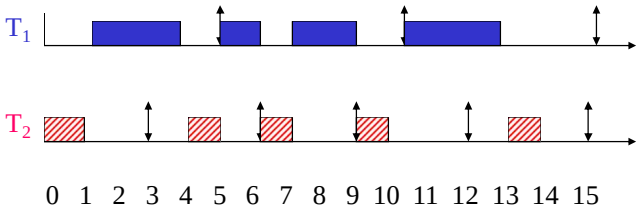
CNS pour les systèmes à échéance sur requête :

$U \leq 1$

EDF : exemple

$r_1 = r_2 = 0$ $D_1 = P_1 = 5$ $D_2 = P_2 = 3$

$C_1 = 3$ $C_2 = 1$



$U = 3/5 + 1/3 = 14/15 < 1 \Rightarrow$ le système est ordonnançable par EDF

Serveur sporadique (sporadic server)

- Le serveur sporadique permet d'améliorer les temps de réponse des tâches apériodiques, sans affecter les tâches périodiques.
- Le serveur sporadique ne réapprovisionne pas sa capacité de manière périodique mais seulement quand elle a été consommée par des tâches apériodiques.
- Il existe plusieurs formes de serveurs sporadiques, pour priorités fixes ou dynamiques (les serveurs sporadiques se distinguent par la manière de consommer et réapprovisionner la capacité de service)

Fonctionnement d'un serveur sporadique simple (avec priorités fixes)

PR_{exe} : priorité de la tâche en cours d'exécution.
PR_s : priorité du serveur sporadique.
P_s : période du serveur sporadique. C_s : capacité maximale du serveur sporadique.
***RT* : instant où aura lieu le réapprovisionnement de la capacité du SS.**
***RA* : quantité de réapprovisionnement qui sera rajoutée à la capacité du SS.**
Le SS est dit activable quand $PR_{exe} \geq PR_s$ (SS en cours de service ou bloqué par une tâche plus prioritaire). Le SS est dit inactif quand $PR_{exe} < PR_s$.

Quand une tâche apériodique arrive, elle s'exécute si la capacité du serveur n'est pas nulle et si la priorité du serveur le permet (les tâches apériodiques s'exécutent avec le même niveau de priorité que le serveur sporadique).

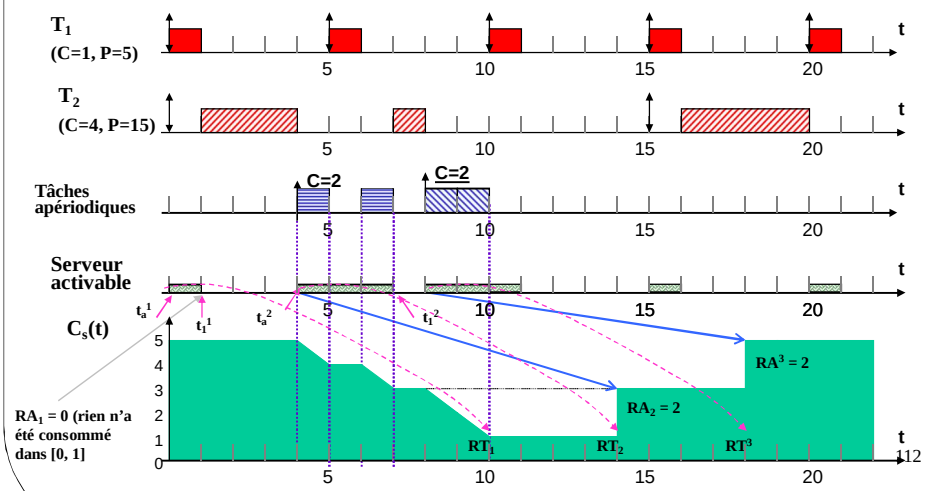
Fonctionnement d'un serveur sporadique simple (avec priorités fixes)

Règles de réapprovisionnement de la capacité du serveur sporadique

- A l'initialisation, la capacité du SS est approvisionnée avec la valeur maximale C_s.
- L'instant de réapprovisionnement RT est fixé dès le moment où le SS devient activable et $C_s > 0$ (soit t_a ce moment). La valeur de RT est fixé à $t_a + P_s$.
- La quantité de réapprovisionnement, RA, à effectuer à l'instant RT est fixée au moment où le SS devient inactif ou quand sa capacité est épuisée (soit t_1 ce moment). La valeur de RA est fixée à une quantité égale à celle consommée durant l'intervalle $[t_a, t_1]$.

Serveur sporadique avec RM : exemple (1)

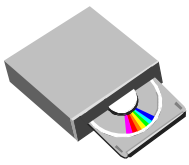
T_s ($r_0 = 0$, $C_s = 5$, $P_s = 10$, $D_s = 10$)



Serveur sporadique avec RM : exemple

- A $t = 0$, T_1 (tâche la plus prioritaire) est activée et le SS (serveur sporadique) devient activable. Comme $C_s > 0$, RT_1 est fixé à $t + P_s = 10$. ($t_a^1 = 0$)
- A $t = 1$, T_1 se termine, le SS devient inactif, car il n'a pas de tâche apériodique à servir. RA_1 (pour $t_1^1 = 1$) est égale à 0 et aucun réapprovisionnement n'est effectué à l'instant RT_1, car aucune quantité de la capacité du SS n'a été consommée dans l'intervalle [0, 1].
- A $t = 4$, une tâche apériodique, T_3, arrive. Comme $C_s > 0$, le SS devient actif et T_3 s'exécute. Par conséquent un réapprovisionnement est prévu à $RT_2 = t + P_s = 14$. ($t_a^2 = 4$)
- A $t = 5$, la tâche apériodique T_3 est préemptée par T_1. Le SS reste activable pendant la préemption.
- A $t = 6$, T_1 se termine et T_3 reprend son exécution et se termine à $t = 7$.
- A $t = 7$, le SS devient inactif. La quantité de réapprovisionnement à faire à RT_2 est égale à la capacité consommée pendant l'intervalle [4, 7], c'est-à-dire, $RA_2 = 2$.
- A $t = 8$, le SS devient actif pour servir une nouvelle tâche apériodique, T_4, arrivée à $t = 8$. Un nouveau réapprovisionnement est prévu à $RT_3 = t + P_s = 18$. ($t_a^3 = 8$)
- A $t = 10$, T_4 se termine, mais le SS reste activable car $PR_{exe} \geq PR_s$.
- A $t = 11$, le SS devient inactif et la quantité de réapprovisionnement à faire à l'instant RT_3 est fixée à $RA_3 = 2$ (2 étant la capacité consommée dans l'intervalle [8, 11]).

Partage de ressources



Fichiers, BD

Utilisation de ressources critiques

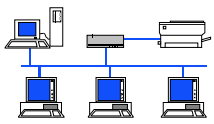
- **Mono ou multi exemplaires**
- **Mode lecture/écriture**
- **Demande de 1 ou n ressources**

Ressource physique

E/S

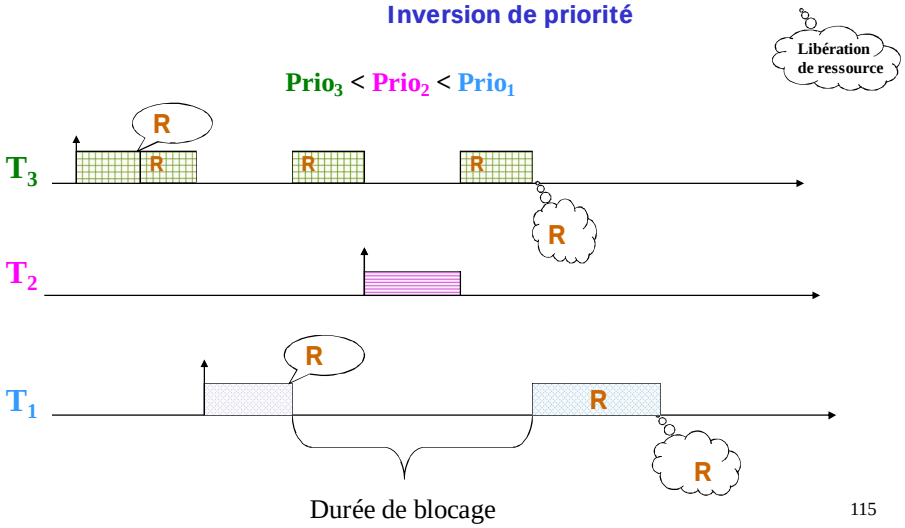
Difficultés :

- **Interblocage**
- **Inversion de priorités**



Réseau

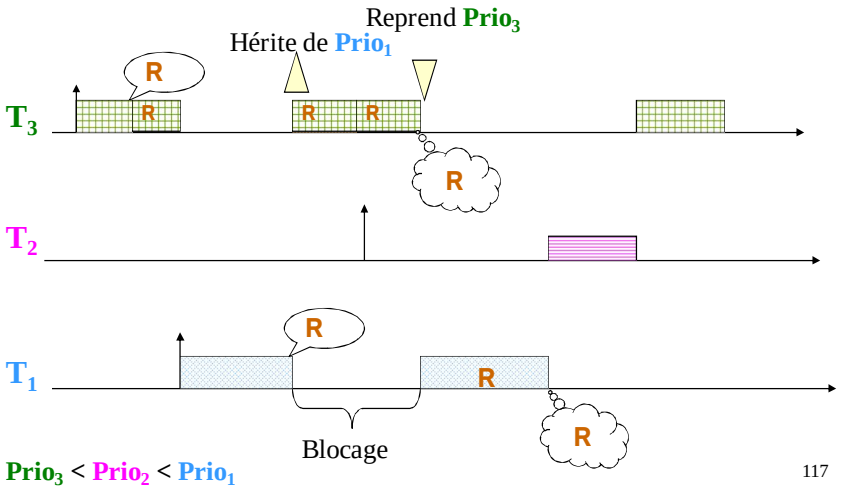
Inversion de priorité



Protocole à héritage de priorité : PIP (Priority Inheritance Protocol)

- **Principe**
  - La tâche T_i qui possède une section critique prend une priorité égale au Max{$Prio_i$, $Prio_k$ | T_k en attente de la section critique}.
  - Quand T_i sort de la SC, elle reprend sa priorité initiale.
- **Utilisable pour les ressources mono exemplaires**

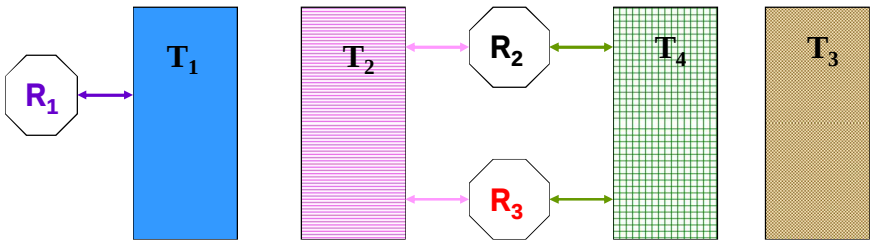
Exemple du protocole à héritage de priorité



Protocole à plafond de priorité : PCP (Priority Ceiling Protocol)

- **Principe**
  - Chaque ressource R_x possède une priorité plafond égale à $Max(Prio_k | T_k$ pouvant accéder à $R_x\}$.
  - On note Π_s le maximum des priorités plafonds des ressources en cours d'utilisation par des tâches. Π_s est égal à une valeur inférieure à la priorité la plus basse quand aucune ressource n'est utilisée.
  - Quand une tâche T_i veut accéder à une ressource R_x qui est libre :
    - Cas 1 : si $Prio_i$ est supérieur à Π_s, alors T_i obtient R_x.
    - Cas 2 : si $Prio_i$ n'est pas supérieur à Π_s, alors T_i obtient R_x seulement si T_i est la tâche qui détient la (ou les) ressource(s) dont le plafond est égal à Π_s.
    - Autres cas : T_i est bloquée.
  - Une tâche T_i qui est en SC prend la priorité $Prio = Max\{Prio_i, Prio_k | T_k$ en attente de SC$\}$. En d'autres termes, si une tâche T_i bloque une tâche T_j, T_i hérite de $Prio_j$ si $Prio_j > Prio_i$. Quand T_i sort de la SC, elle reprend la priorité quelle avait avant d'entrer en SC.

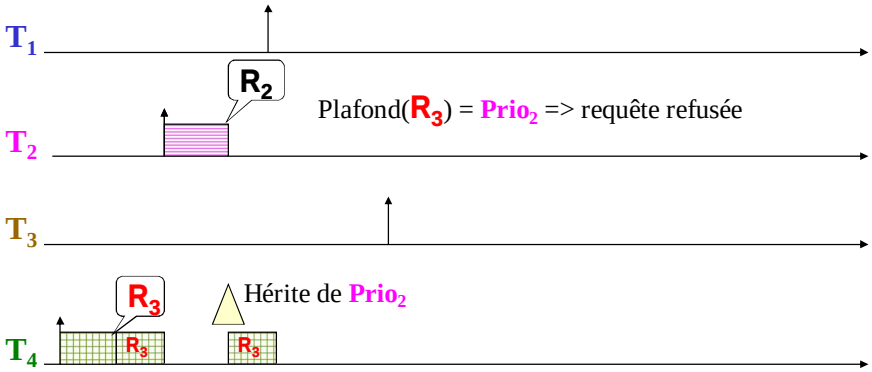
Exemple de fonctionnement de PCP



$Prio_4 < Prio_3 < Prio_2 < Prio_1$

Plafond(R_1) = $Prio_1$ Plafond(R_2) = $Prio_2$ Plafond(R_3) = $Prio_2$

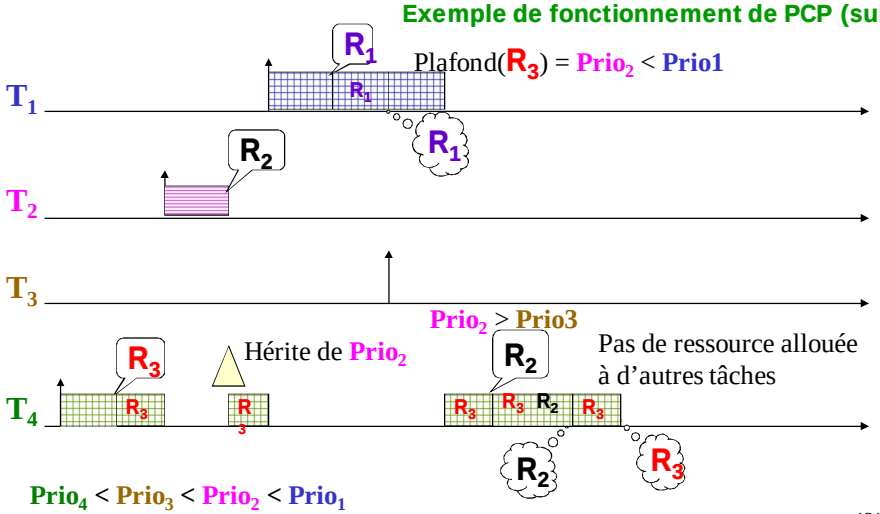
Exemple de fonctionnement de PCP (suite)



$Prio_4 < Prio_3 < Prio_2 < Prio_1$

Plafond(R_1) = $Prio_1$ Plafond(R_2) = $Prio_2$ Plafond(R_3) = $Prio_2$

Exemple de fonctionnement de PCP (suite)

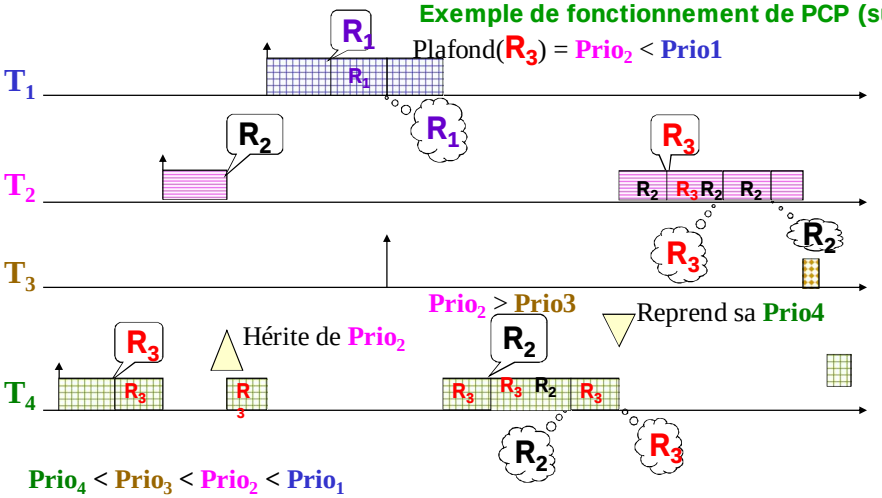


$Prio_4 < Prio_3 < Prio_2 < Prio_1$

Plafond(R_1) = $Prio_1$ Plafond(R_2) = $Prio_2$ Plafond(R_3) = $Prio_2$

121

Exemple de fonctionnement de PCP (suite)



$Prio_4 < Prio_3 < Prio_2 < Prio_1$

Plafond(R_1) = $Prio_1$ Plafond(R_2) = $Prio_2$ Plafond(R_3) = $Prio_2$

122

3. Ordonnancement avec POSIX

- Principes

- Chaque processus peut demander à être ordonnancé avec une politique qu'il choisit.

- Il y a quatre politiques :
  * SCHED_FIFO
  * SCHED_RR
  * SCHED_SPORADIC (introduit par Posix de 1999)
  * SCHED_OTHER (attention à la portabilité des applications!)

- Chaque politique définit un nombre minimum de priorités (32 niveaux au moins) pour ordonnancer les processus.

- Posix utilise une structure `sched_param` qui contient les infos nécessaires à l'ordonnancement pour chaque politique supportée.

- L'utilisation des primitives d'ordonnancement nécessite l'intégration du fichier entête `<sched.h>`

- Quand des threads sont en attente sur des *mutex* ou variables conditionnelles, ils sont réveillés selon leur ordre de priorité (i.e. le plus prioritaire est réveillé en premier).

3. Ordonnancement avec POSIX

- SCHED_SPORADIC

- Technique utilisée pour servir des threads apériodiques.
- Un **serveur sporadique** est défini par deux paramètres (contenu dans `sched_param`)
  * la période de réapprovisionnement du serveur (`sched_ss_repl_period`)
  * capacité du serveur (initialisée à `sched_ss_init_budget`)

- La priorité d'un thread servi par la politique SCHED_SPORADIC commute entre les valeurs de `sched_priority` et `sched_ss_low_priority` de la structure `sched_param`

- La priorité affectée au thread servi par SCHED_SPORADIC est définie comme suit :
  * si la capacité restante du Serveur Sporadique n'est pas nulle et le nombre d'opérations de réapprovisionnement est inférieur à `sched_ss_max_repl`, la priorité du thread est fixée à `sched_priority`. Sinon, elle est fixée `sched_ss_low_priority`.

  * si la valeur de `sched_priority` est inférieure ou égale à celle de `sched_ss_low_priority`, le fonctionnement du système est indéfini. Il s'agit d'une mauvaise utilisation des priorités.

- SCHED_SPORADIC (suite)

La capacité disponible du SS est modifiée selon les règles suivantes :

(1) Quand le thread en tête de file d'attente associée à la priorité sched_priority devient actif, son temps d'exécution ne doit pas dépasser la capacité disponible du SS.

(2) A chaque fois que le thread est recyclé, c'est-à-dire mis en queue de la file associée à la priorité `sched_priority`, parce que le thread est bloqué ou parce qu'une opération de réapprovisionnement a eu lieu, l'instant où cette opération est effectuée est mémorisée dans `activation_time`.

(3) Quand le thread en exécution, avec la priorité égale à `sched_priority`, est préempté, il est mis en tête de la liste associée à cette priorité et le temps qu'il a consommé est soustrait de la capacité disponible.

(4) Quand le thread en exécution, avec la priorité égale à `sched_priority`, est bloqué le temps consommé est soustrait de la capacité disponible et l'opération de réapprovisionnement est programmée comme l'indiquent les règles (6) et (7).

- SCHED_SPORADIC (suite)

(5) Quand le thread en exécution, avec la priorité égale à `sched_priority`, atteint la limite imposée par son temps d'exécution, il est mis en queue de liste associée à la priorité `sched_ss_low_priority`. Le temps consommé est soustrait de la capacité disponible et l'opération de réapprovisionnement est programmée comme l'indiquent les règles (6) et (7).

(6) Chaque fois que l'opération de réapprovisionnement a lieu, la quantité de réapprovisionnement, notée `replenish_amount`, est égale au temps consommé depuis l'instant `activation_time` du thread. Le réapprovisionnement est programmé à l'instant `activation_time` plus la valeur de `sched_ss_repl_period`. On peut avoir plusieurs opérations de réapprovisionnement (sans dépasser `sched_ss_max_repl` opérations) qui sont pendantes en même temps, chacune étant servie à son ton instant respectif.

(7) Une opération de réapprovisionnement consiste à ajouter la quantité `replenish_amount` à la capacité disponible au moment du réapprovisionnement. La valeur de la capacité disponible est ramenée `sched_ss_initial_budget` si le réapprovisionnement conduit à dépasser ce seuil.

- Combinaison de politiques

  - Un système peut gérer plusieurs processus avec des politiques différentes.

Tâche	Politique	Priorité	Quantum
T1	SCHED_FIFO	**1**	-
T2	SCHED_FIFO	**2**	-
T3	SCHED_OTHER	**-**	-
T4	SCHED_OTHER	**-**	-
T5	SCHED_RR	**5**	15
T6	SCHED_RR	**5**	5
T7	SCHED_RR	**5**	10
T8	SCHED_FIFO	**3**	-
T9	SCHED_FIFO	4	-
T10	SCHED_SPORADIC	**6**	-
T11	SCHED_SPORADIC	**7**	-

API pour l'ordonnancement

- Primitives pour la manipulation des paramètres et politique d'ordonnancement

  - La primitive `pthread_getschedparam` permet d'obtenir les paramètres d'ordo d'un thread

```
int pthread_getschedparam (pthread_t thread, int *policy,
        struct sched_param *param)
/* sched_param contient (en général) la priorité.
```

  - La primitive `pthread_setschedparam` permet de modifier les paramètres d'ordo d'un thread

```
int pthread_setschedparam (pthread_t thread, int policy,
        const struct sched_param *param)
```

  - La primitive `sched_setscheduler` permet de modifier la politique d'ordonnancement et les paramètres (i.e. la priorité) d'ordo d'un processus

```
int sched_setsheduler (pid_t pid, int policy,
        const struct sched_param *param)
```

  - La primitive `sched_getscheduler` permet d'obtenir la politique d'ordonnancement d'un processus

```
int sched_getscheduler (pid_t pid)
```

- **Primitives pour la manipulation des paramètres et politique d'ordonnancement** (suite)

- La primitive `sched_get_priority_max` permet d'obtenir le niveau de priorité le plus élevé d'une politique d'ordonnancement donnée.

```
int sched_get_priority_max (int policy)
```

- La primitive `sched_get_priority_min` permet d'obtenir le niveau de priorité le plus bas d'une politique d'ordonnancement donnée.

```
int sched_get_priority_min (int policy)
```

- La primitive `sched_rr_get_interval` permet d'obtenir la valeur du quantum associé à l'exécution par round robin du processus spécifié.

```
int sched_rr_get_interval (pid_t pid, struct timespec *interval)
```

- La primitive `sched_yield` permet au processus appelant de libérer le processeur (CPU) et de se mettre en queue de file (i.e. il cède volontairement le processeur).

```
int sched_yield (void)
```


- Primitives de manipulation des attributs liés à l'ordonnancement

- Modifier les attributs des threads revient à remplir la structure des attributs de threads `attr` qui est du type **`pthread_attr_t`**.

- La primitive `pthread_attr_init` permet d'initialiser la structure d'attributs d'un thread et la remplit avec les valeurs par défaut pour tous les attributs

```
int pthread_attr_init (pthread_attr_t *attr)
```

Valeurs par défaut (ces valeurs peuvent être changées par les primitives `pthread_attr_set<nom de l'attribut>`):

* Etat de création avec `PTHREAD_CREATE_JOINABLE` (qui signifie qu'un autre thread peut se synchroniser avec la fin du thread et reprendre son exécution à la fin du thread en utilisant `pthread_join`)
* Politique d'ordonnancement = SCHED_OTHER
* Priorité = 0
* Etat de création avec PTHREAD_EXPLICIT_SCHED (qui signifie que le thread créé hérite des attributs d'ordonnancement de son père).

- La primitive `pthread_attr_destroy` permet de supprimer des attributs d'un thread

```
int pthread_attr_destroy (pthread_attr_t *attr)
```

- Primitives de manipulation des attributs liés à l'ordonnancement (suite)

- La primitive `pthread_attr_getschedparam` permet de connaître les attributs d'ordonnancement d'un thread.

```
int pthread_attr_getschedparam (const pthread_attr_t *attr
        struct sched_param *param)
```

- La primitive `pthread_attr_setschedparam` permet de modifier les attributs d'ordonnancement d'un thread.

```
int pthread_attr_setschedparam (pthread_attr_t *attr
        const struct sched_param *param)
```

- La primitive `pthread_attr_getschedpolicy` permet de connaître la politique d'ordonnancement d'un thread.

```
int pthread_attr_getschedpolicy (const pthread_attr_t *attr, int *policy)
```

- La primitive `pthread_attr_setschedpolicy` permet de modifier la politique d'ordonnancement d'un thread.

```
int pthread_attr_setschedpolicy (pthread_attr_t *attr, int policy)
```


- Primitives de manipulation des attributs liés à l'ordonnancement (suite)

- La primitive `pthread_attr_getinheritshed` permet de connaître la valeur de l'attribut d'héritage des propriétés d'ordonnancement du fils à partir du père.

```
int pthread_attr_getinheritsched (const pthread_attr_t *attr,int *inheritsched)
```

- La primitive `pthread_attr_setinheritsched` permet de modifier d'attribut d'héritage des propriétés d'ordonnancement

```
int pthread_attr_setinheritsched (pthread_attr_t *attr, int inheritsched)
```

- La primitive `pthread_attr_getscope` permet de connaître l'étendue des propriétés d'ordonnancement d'un thread (pour le moment seul l'étendue *Système global* est spécifiée)

```
int pthread_attr_getscope (const pthread_attr_t *attr, int *scope)
```

- La primitive `pthread_attr_setscope` permet de modifier l'étendue des propriétés d'ordonnancement d'un thread.

```
int pthread_attr_setscope (pthread_attr_t *attr, int scope)
```

Utilisation de *mutex* à priorité

- Inversion de priorité : problème crucial pour les STRE
- Mutex à plafond de priorité : une solution pour éviter l'inversion de priorité

- Si `defined(_POSIXTHREAD_PRIO_PROTECT) || defined(_POSIX_THREAD_PRIO_INHERIT)` alors on peut utiliser les primitives sur les mutex à priorité.

  * Si la condition `defined(_POSIXTHREAD_PRIO_PROTECT)` est satisfaite, le système support le **protocole à plafond de priorité** ('Priority ceiling')
  * Si la condition `defined(_POSIX_THREAD_PRIO_INHERIT)` est satisfaite, le système supporte **l'héritage de priorité** ('priority inheritance')

- Définition et fonctionnement de mutex à plafond de priorité

- Au moment de la création du mutex, on spécifie un plafond de priorité. Tout thread qui verrouille le mutex hérite automatiquement de cette priorité, ce qui lui permet de s'exécuter sans être interrompu par un autre thread qui tenterait de verrouiller le mutex.

- On peut lire ou modifier les attributs d'un mutex à plafond de priorité.

- **Problèmes** :
  * Les mutex à priorités ont un surcoût plus élevé que les mutex normaux.
  * S'il y a des threads qui ont une priorité plus élevée que celle du mutex, le protocole évitement d'inversion de priorité peut devenir inefficace

- Primitives de manipulation de mutex à plafond de priorité

- La primitive `pthread_mutexattr_getprioceiling` permet de connaître la priorité d'exécution d'un thread (créé avec pthread_mutexattr_init) après qu'il ait verrouillé un mutex.

```
int pthread_mutexattr_getprioceiling (const pthread_mutexattr_t *attr,
        int *prioceiling)
```

- La primitive `pthread_mutexattr_setprioceiling` permet de spécifier la priorité d'exécution d'un thread (créé avec `pthread_mutexattr_init`) quand celui-ci verrouille un mutex.

```
int pthread_mutexattr_setprioceiling (pthread_mutexattr_t *attr,
        int prioceiling)
```

- La primitive pthread_mutex_getprioceiling permet de connaître la priorité d'exécution d'un thread après qu'il ait verrouillé un mutex.

```
int pthread_mutex_getprioceiling (const pthread_mutex_t *mutex,
        int *prioceiling)
```

- La primitive `pthread_mutex_setprioceiling` permet spécifier la priorité d'exécution d'un thread quand celui-ci verrouille un mutex

```
int pthread_mutex_setprioceiling (const pthread_mutex_t *mutex,
        int prioceiling, int *old_ceiling)
```


- Définition et fonctionnement de mutex à héritage de priorité

- Quand un thread verrouille un mutex, la priorité des threads qui demandent à verrouiller le mutex est contrôlée ainsi :

  1) Si un thread a verrouillé un mutex, il hérite de la priorité la plus élevée des threads bloqués en attente de ce mutex, à condition que sa priorité soit inférieure à celles des demandeurs.

  2) Quand un thread déverrouille un mutex, il reprend sa priorité initiale.

- On peut lire ou modifier les attributs d'un mutex à héritage de priorité.

- Le protocole à héritage de priorité est le plus répandu

- Primitives de manipulation de mutex à héritage de priorité

- La primitive `pthread_mutexattr_getprotocol` permet de savoir si un mutex (créé avec pthread_mutexattr_init) dispose ou non de l'héritage de priorité.

```
int pthread_mutexattr_getprotocol (const pthread_mutexattr_t *attr,
            int *protocol)
```

- La primitive `pthread_mutexattr_setprotocol` permet de valider ou invalider le protocole d'héritage de priorité pour un mutex (créé avec `pthread_mutexattr_init`)

```
int pthread_mutexattr_setprotocol (pthread_mutexattr_t *attr,
            int protocol)
```

où `protocol` peut prendre l'une des valeurs suivantes :

```
PTHREAD_PRIO_NONE       /* Aucun protocole n'est associé au mutex
PTHREAD_PRIO_INHERIT    /* Protocole d'héritage de priorité associé au mutex
PTHREAD_PRIO_PROTECT    /* Protocole à plafond de priorité associé au mutex
```


Points forts / Points faibles de POSIX

+ Facilite la portabilité des applications
+ Offre un bon support pour l'ordonnancement à priorités fixes

- Ne supporte pas l'ordonnancement dynamique et pas d'activation périodique
- Les intervalles de valeurs des priorités ne sont pas fixés
- Si les intervalles de valeurs de priorités ne sont pas disjoints entre les politiques, que se passe-t-il ?
- La valeur du quantum pour le Round Robin et la possibilité de le changer dépend de l'OS
- Aucune garantie absolue sur les performances temps réel (les appels systèmes sont préemptibles).