Spécification d'un RTOS

Etienne Brodu, Martin Richard, Maxime Gaudin,
Monica Golumbeanu, Paul Adenot, Yoann Rodière

6 décembre 2010

Table des matières

1 Sémaphore a compte

1.1 Définition de l'objet

Le sémaphore a compte est un objet permettant de protéger une ressource critique. On peut représenter le sémaphore par analogie avec une boite qui contient un certain nombre de jetons. Des utilisateurs se partageant une ressource ne peuvent y accéder qu'à la condition de *prendre* un jeton disponible, et le *rendent* une fois qu'ils n'ont plus besoin de la ressource critique.

1.2 Attributs de l'objet

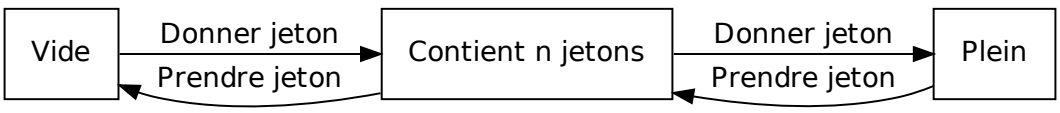
- Capacité (nombre de jetons qu'il peut contenir) `int`
- Etat (nombre de jetons qu'il contient à un instant t) `int`

Le sémaphore est représenté et manipulé par l'utilisateur grâce à un identifiant communiqué à l'utilisateur lors de l'initialisation de l'objet (symbole de type `semid_t`).

1.3 Spécification des primitives

- `semid_t creerSemaphore(int maxJetons, int initJetons)` L'initialisation d'un sémaphore permet de créer l'objet. L'utilisateur définit le nombre de jetons qu'il peut contenir et le nombre qu'il contient à la création de l'objet. `maxJeton` correspond au maximum de jetons que peux accueillir le sémaphore et `initJetons` le nombre de jeton initial.
- `void supprSemaphore(semid_t semId)` La déstruction d'un sémaphore détruit l'objet et le supprime de la mémoire. Si des utilisateurs attendaient la libération d'un jeton, alors ils seront libérés et l'utilisateur sera notifié par un code retour particulier (voir ci-dessous).
- `int prendre(semid_t semId, int attendre)` L'utilisateur souhaitant accéder à la ressource partagée doit prendre un jeton dans le sémaphore. Il peut spécifier la stratégie d'attente si aucun jeton n'est disponible. La fonction retourne 1 si l'utilisateur a obtenu un jeton, 0 si non et -1 si le sémaphore identifié par semId n'existe pas. le paramètre `attendre` vaut 0 si l'utilisateur ne souhaite pas attendre, -1 pour attendre indéfiniment ou tout autre valeur correspondant à la durée d'attente en millisecondes.
- `int donner(semid_t semId)` L'utilisateur souhaite déposer un jeton dans le sémaphore. Les codes retour de la fonction sont les suivants : 1 si le jeton a été rendu, 0 si le sémaphore est plein, -1 si le sémaphore identifié par semId n'existe pas. Ce mécanisme ne vérifie pas si un utilisateur a pris un jeton avec `prendre()` avant de le donner avec `donner()`.

1.4 Diagramme d'état



1.5 Divers

2 Evénement

2.1 Définition de l'objet

Un objet de type événement mémorise le signal marquant qu'une condition est devenue vraie. Il permet la signalisation entre deux tâches - cela implique qu'il y a une tâche émettrice et une tâche destinataire.

2.2 Attributs de l'objet

1. Un événement est spécifique à une tâche, susceptible de l'attendre. Une tâche est en attente du signal alors qu'une autre sera chargée de l'envoyer. On a donc un producteur et un consommateur.
2. Une tâche dispose d'un nombre fini d'événements qu'elle peut attendre.
3. L'association des événements aux tâches ne nécessite aucune file d'attente : elle fournit donc un mécanisme élémentaire ce qui permet de construire des mécanismes de synchronisation de haut niveau et rend l'objet très efficace.
4. L'objet événement permet de gérer la signalisation simple (attente d'un unique évènement) mais aussi certains cas de signalisation multiple : attente d'un évènement parmi plusieurs (séparés par des OU) ou de plusieurs (séparés par des ET). Les cas plus complexes de signalisation multiple (tels la combinaison de ET et de OU) peuvent être gérés en implémentant une agence dédiée.
5. En combinant l'objet événement à l'agence Horlogerie on pourrait gérer des attentes avec délai comme des time-out.

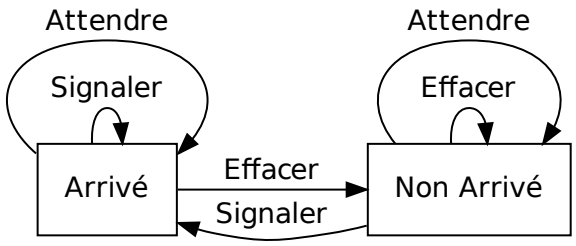
2.3 Spécification des primitives

1. **boolean ARRIVE (listeEvenements, operateur)**
   - Utile pour la signalisation multiple, cette primitive permet de vérifier si les éléments de la liste ListeEvenements sont arrivés en tenant compte s'ils sont séparés par des opérateurs OU ou ET (les valeurs possibles du paramètre *opérateur*).
2. **void ATTENDRE (listeEvenements, operateur)**
   - La tâche appelante se met dans l'état *en attente* (sous le contrôle de l'ordonnanceur) si les événements de la liste *listeEvenements* ne sont pas arrivés. Le paramètre *operateur* peut prendre les valeurs ET ou OU et permet de spécifier si tous les événements de la liste *listeEvenements* sont attendus ou juste un seul d'entre eux. Dès que les événements nécéssaires sont arrivés, la tâche passe à l'état *prêt* et on appelle l'ordonnaceur.
3. **void EFFACER (listeEvenements)**
   - Tous les événements de la liste *listeEvenements* sont mis dans l'état *NON ARRIVé*.
4. **void SIGNALER (idEvenement, idTache)**
   - L'événement qui a l'identifiant *idEvenement* est mis dans l'état *ARRIVé* quelque soit sont état initial. Si la tâche réceptrice est dans l'état *en attente* et qu'elle n'attend plus d'événement, elle passe à l'état *prêt* et l'ordonnanceur est invoqué. Sinon, l'arrivée de l'évènement est mémorisée et l'état de la tâche ne change pas.

2.4 Diagramme d'état

Les deux états dans lesquels un événement peut se trouver sont *NON ARRIVé* et *ARRIVé*.

2.5 Divers



3 Mémoire partagé

3.1 Définition de l'objet

Un segment de mémoire partagée permet de partager des informations, dans un bloc mémoire, entre plusieurs tâches. Aucun mécanisme d'exclusion mutuelle n'est géré par la mémoire partagée, qui doit donc être protégée de manière extérieur si besoin est.

La mémoire partagée peut être accédée comme un fichier. Ce fichier peut (et sera dans la plupart des cas) associé en mémoire par un appel de type `mmap`.

Le Task Control Block contient un descripteur de fichier (petit entier positif) associé à la mémoire partagée.

3.2 Attributs de l'objet

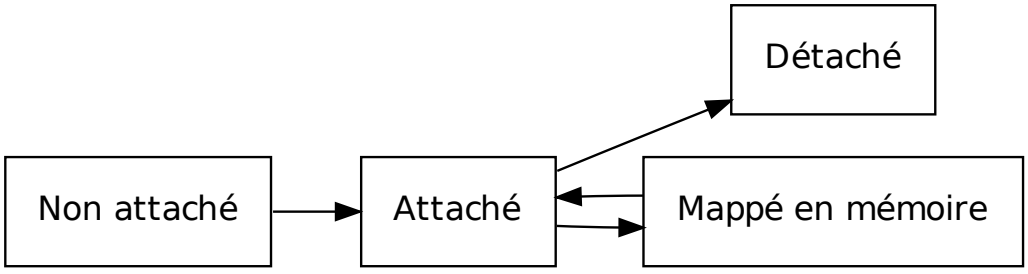
3.3 Spécification des primitives

`int mempart_init(nom, mode) :` ouvre un segment de mémoire partagé représenté dans le système par `nom`, avec le mode d'ouverture `mode` (création, lecture, écriture, etc.).

`int mempart_fermer(fd) :` ferme le segment de mémoire partagée, en vidant les buffers.

L'attachement du fichier en mémoire ne fait pas strictement partie de l'API de cette tâche.

3.4 Diagramme d'état



3.5 Divers

4 Région

4.1 Définition de l'objet

Les objets « région »assurent la gestion des exclusions mutuelles (*mutex*), C'est à dire que la tâche utilisant cette région ne sera jamais préemptée, pas même par des tâches de priorité plus haute.

Les objets de type région peuvent prendre deux états : LIBRE ou OCCUPÉE. À sa création, une région est libre.

4.2 Attributs de l'objet

– Identifiant : `int` $\in [0, 2^{16} - 1]$

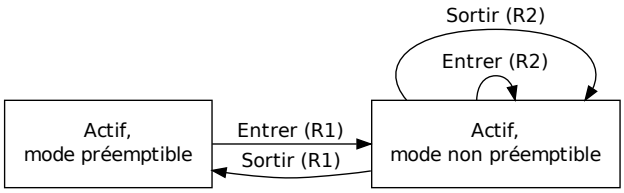
4.3 Spécification des primitives

Toutes les primitives renvoient -1 en cas d'erreur (*e.g* plus de mémoire disponible, identifiant invalide, *etc.*). Sauf indication contraire, elles renvoient 0 en cas de succès.

– `int entrer(int id)` : Rentre dans le région dont l'identifiant est passé en paramètre. Dans un contexte monoprocesseur, on est assuré que si la région est déjà occupée, elle l'est par la tâche courante. Dans le cas où la tâche demande à rentrer dans un région dans laquelle elle est déjà, sa demande est ignorée.

– `int sortir(int id)` : Sort de la région dont l'identifiant est passé en paramètre. Si une tâche demande à sortir d'une région dans laquelle elle n'est pas, sa demande est ignorée.

4.4 Diagramme d'état



4.5 Divers

Il est à noter que :

– Les opérations d'attente passive et de terminaison de la tâche sont interdites dans une région.

– Les régions sont imbricables les unes dans les autres.

– Dans un contexte multi-processeur, il peut y avoir concurrence d'accès à la région entre deux tâche. Dans ce cas, la primitive `entrer` est bloquante pour la deuxième tâche appelante tant que la région n'est pas libérée.

5 Tâche

5.1 Définition de l'objet

Une tâche représente un fil d'exécution, *i.e.* un ensemble d'instructions exécuté sur un même processeur. Elle peut s'exécuter (état `ACTIVE`), attendre d'avoir la main sur le processeur (état `ACTIVABLE`) ou attendre l'autorisation d'une ou plusieurs autres tâches.

Si une des méthode `attendre`, `entrer`, `suspendre`, ou `prendre` est appelée, la tâche appelante passe dans le méta-état `BLOQUE` jusqu'à ce que l'une des méthode suivante soit appelée : `signaler`, `sortir`, `reprendre` ou `donner`.

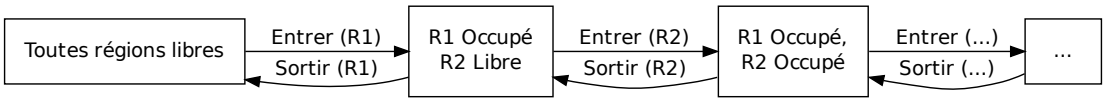
5.2 Attributs de l'objet

La structure `struct tache` est composée de :
- id : `int`
- etat : `enum { ACTIVABLE, ACTIVE, SUSPENDUE, ATTENTE_EVENEMENT, ATTENTE_REGION, ATTENTE_SEMAPHORE }`
- attente_id : `int`
- evenements_attente : Liste d'identifiants d'événements.
- priorite : `int`

5.3 Spécification des primitives

Toutes les primitives avec un type de retour entier, renvoient -1 en cas d'erreur (*e.g* plus de mémoire disponible, identifiant invalide, *etc.*). Sauf indication contraire, elles renvoient 0 en cas de succès.
- `int demarrer(struct tache* t)` : Démarre la tâche passée en paramètre.
- `int suspendre(struct tache* t)` : Suspend la tâche passée en paramètre.
- `int terminer(struct tache* t)` : Tue la tâche passée en paramètre.
- `struct tache* tacheCourrante()` : Renvoie la structure de la tâche courante, *i.e.* la tâche d'où cette fonction est appelée.

5.4 Diagramme d'état



5.5 Divers

L'ordonnancement des tâches est délégué à une agence.

6 Agence horlogerie

6.1 Définition de l'agence

L'agence horlogerie fournie un service d'acquisition du temps. Elle offre également la création d'évenement à une date ou après l'écoulement d'un temps.

6.2 Structures de l'agence

- La structure `struct duree_t` est composée de :
  - seconde : `int`
  - mseconde : `int`
- La structure `struct time_t` est composée de :
  - heure : `int`
  - minute : `int`
  - seconde : `int`
  - mseconde : `int`
- La structure `struct date_t` est composée de :
  - jour : `int`
  - mois : `int`
  - annee : `int`
- La structure `struct datetime_t` est composée de :
  - date : `struct date_t`
  - time : `struct time_t`

6.3 Spécification des primitives

- `time_t getCurrentTime()` Cette primitive permet de récupérer le temps courant.
- `date_t getCurrentDate()` Cette primitive permet de récupérer la date courante.
- `setCurrentTime(time_t time)` Cette primitive permet de définir le temps courant.
- `setCurrentDate(date_t date)` Cette primitive permet de définir la date courante.
- `event_id createEvent (datetime_t dateTime)` Créer un évenment qui sera déclenché à la date `dateTime`.
- `event_id createEvent (duree_t duree)` Créer un évenement qui sera déclenché à la date courante + `duree`.

6.4 Divers

6.4.1 Reflexion sur le rapport entre f1 et f2

- *f1* : fréquence des interruptions de l'horloge temps réel.
- *f2* : fréquence des temps minimum d'execution d'une tache avant préemption

La fréquence *f1* est la fréquence d'execution des instruction cadencé par l'horloge temps réel. Il n'est pas avantageux d'avoir une fréquence *f2* trop proche de la fréquence d'execution des instructions, car le changement de contexte pourrais être alors plus consommateur en ressources que l'execution des tâches elle-même.

Plus *f2* est proche de *f1*, meilleur est le temps de réponse, car les tâches rendent la main très fréquement. Il sera donc rare d'attendre une tâche longtemps. En revanche, le temps d'execution sera rallongé du temps de changement de contexte fréquent.

Plus *f2* est éloigné de *f1*, meilleur est l'efficacité d'execution. On économise tout les temps de passage d'une tâche à l'autre, en revanche, la réactivité est moins bonne.

Le temps moyen d'execution d'une primitive seras 10 à 100 fois plus long que le temps d'éxecution d'une instruction, car les primitives comportent plusieurs dizaine d'instructions. Le temps moyen d'execution d'une tâche doit permettre à plusieurs primitive de s'executer, il sera donc environ 100 fois plus long que le temps d'execution d'une primitive.