# Final TR52

Parties 1 et 2 sur feuilles séparées SVP

Automne 2007

# 1 Les exécutifs temps réel (14 points)

## 1.1 Les événements logiciels

On souhaite mettre en place le service des événements logiciels, caractérisé par les primitives :

```
EVTID evtCreate()
int evtSignal(EVTID evId)
int evtWait(EVTID evId)
```

Où `EVTID` est le type "identité d'événement" (un pointeur vers un descripteur d'événement). L'appel de la fonction `evtWait`, avec une identité *id* d'événement provoque systématiquement la mise en attente de la tâche appelante, jusqu'au prochain appel de la fonction `evtSignal`, avec la même identité *id*. L'appel de la fonction `evtSignal` avec l'identité *id* d'évenement fait repasser dans l'état PRETE ***toutes*** les tâches qui sont en attente suite à l'appel de `evtWait` avec la même identité *id*.

1. proposer une structure de données pour le descripteur d'événement logiciel.
2. proposer les algorithmes des fonctions `evtWait` et `evtSignal`, en s'inspirant des algorithmes des fonctions $P$ et $V$ des sémaphores, données en cours.

### 1.2 Analyse d'une application

On réalise, à l'aide d'un exécutif temps-réel préemptif, le programme suivant formé de trois tâches concurrentes `T1`, `T2` et `T3`, de priorités respectives $p_1 > p_2 > p_3$, d'un événement `e1` et d'un sémaphore d'exclusion mutuelle `S` :

```
void T1(void){ evWait(e1) ; <t1.1> ; P(S) ; <t1.2> ; V(S) ; }
void T2(void){ evWait(e1) ; <t2.1> ; }
void T3(void){ P(S) ; <t3.1> ; evSignal(e1) <t3.2> ; V(S) ; }
main() {
tskCreate(T1,p1, ...) ;
tskCreate(T2,p2, ...) ;
tskCreate(T3,p3, ...) ;
while(1) { }
}
```

Les parties notées `<ti.j>` correspondent à des sections de code non détaillées. Donner un diagramme temporel d'exécution (chronogramme) pour ce programme. On fera en particulier figurer à tout instant l'état de chaque tâche (active, prête, bloquée sur sémaphore, sur événement, ...) et la valeur du sémaphore S. On indiquera aussi clairement les principaux événements (réveil d'une tâche, prise de sémaphore, ...). La procédure `main()` est traitée par l'exécutif comme étant la tâche de fond, qui possède la priorité la plus basse. Que se passe-t-il si la durée d'exécution de la section `<t2.1>` est importante ? Ce résultat vous semble-t-il compatible avec le fait qu'une tâche `T` de priorité donnée ne peut être retardée que par une tâche de priorité supérieure ou une tâche accédant à une ressource critique utilisée par cette tâche `T` ? Expliquez ce qui produit ici et proposez éventuellement une solution pour pallier ce problème.

## 2 Questions java temps réel (6 points)

Un robot mobile d'exploration est équipé de moteurs, d'une caméra et d'un capteur de contact à l'avant. Le système d'exploitation installé sur le robot supporte les programmes écrits en java temps réel. Le contrôle du robot est constitué de plusieurs tâches.

## 2.1 Tâche d'acquisition

Une première tâche intitulée "Acquisition" devra acquérir et traiter les images perçues par la caméra, toutes les secondes. Pour capturer une image de la caméra, on utilisera la méthode "getImage()" de la classe "Camera" qui renvoie un objet de type "Image". Les classes "Camera" et "Image" sont données et n'ont pas besoin d'être écrites. A partir de cette image, on pourra obtenir la commande à appliquer au robot en effectuant un traitement approprié par la méthode "getDestination(Image)" de la classe "TraiteImage", qui renvoie la "Position" à atteindre (de même, la classe "Position" est donnée).

On sait que l'acquisition d'une image par la caméra, ou son traitement prend en moyenne 0,5 s, mais peut dans certaines situations prendre un temps nettement supérieur à une seconde. Dans ce cas, les informations perçues par le robot ne seront plus adaptées dans le cas où celui-ci serait encore en mouvement. C'est pourquoi on décide de réinitialiser cette tâche si elle ne s'est pas terminée au bout d'une seconde.

Donner le code java temps réel de cette tâche en expliquant vos choix.

## 2.2 Tâche de contrôle

Une deuxième tâche "Control" est chargée de contrôler les moteurs du robot pour le déplacer par une poursuite de trajectoire depuis sa position actuelle vers un point d'arrivée. Pour obtenir une finesse suffisante de la trajectoire, on décide de séquencer cette tâche tous les dizièmes de seconde. On dispose pour cela d'une méthode "calculeCommande (Position posarrivee)" qui renvoie la commande (instance de la classe "Commande") à appliquer aux moteurs pour atteindre la position finale "posarrivee". La commande ainsi obtenue peut par la suite être appliquée aux moteurs par l'intermédiaire de la méthode "setCommande(Commande)" de la classe "Moteur". Les classes "Moteur" et "Commande" sont également données.

Pour éviter tout problème, on veut que cette tâche ne puisse pas être interrompue par la tâche "Acquisition", ni par le ramasse-miettes, qui sont des tâches "longues". De plus, si un quelconque retard apparait dans son exécution, on décide d'arrêter les moteurs par la méthode "stop()" de la classe "Moteur" et d'attendre que la tâche d'acquisition puisse déterminer la nouvelle position à atteindre.

Ecrire également le code java de cette classe et expliquer vos principaux choix.

## 2.3 Tâche de surveillance

Finalement, on décide de mettre en place une surveillance périodique du capteur de contact toutes les 10 millisecondes, au travers de la méthode "boolean teste()" de la classe "Capteur". Si celle-ci renvoie "true", c'est qu'il y a eu contact. Dans ce cas, on arrête les moteurs à l'aide de la commande "stop()".

Ecrire le code java de cette classe.