Operateur Select de Ada

Ensimag 2A

1 Ada : select-when/accept-call

En Ada, il existe deux grandes familles de synchronisations intégrées au langage : la construction de moniteurs et l'utilisation de rendez-vous. C'est cette deuxième méthode que nous allons utiliser dans ce problème. Son principe est souvent d'utiliser un thread dédié (un serveur) auquel on envoie des requêtes. Pour donner un exemple, le code pour implanter un sémaphore avec un serveur (Task) et des rendez-vous est le suivant [modifié de `http://en.wikibooks.org/wiki/Ada_Programming/Tasking`, CC-BY-SA] :

```
task type Semaphore_Serveur is
    entry Init (N : in Natural);
    entry Wait;
    entry Post;
end Semaphore_Serveur;

task body Semaphore_Serveur is
   Count : Natural;
begin
   accept Init (N : in Natural) do
      Count := N;
   end Initialize;
   loop
      select
         when Count > 0 =>
              accept Wait do
                 Count := Count - 1;
              end Wait;
      or
              accept Post;
              Count := Count + 1;
      end select;
   end loop;
end Semaphore_Serveur;


SemS : Semaphore_Serveur
...
SemS.Init(3);
...
SemS.Wait;
SemS.Post;
```

Une **Task** est donc une fonction exécutée par un thread dédié.

2 Réimplanter le select de Ada en C

Le but de ce problème est de reconstruire un opérateur de type select-when/accept-call en C pour implanter le même algorithme avec le même paradigme. Pour cela nous utiliserons les fonctions et structures suivantes :

```
struct sel_t;
struct sel_config {
  bool (*fgarde)(void);
  char *message_name;
  void (*faction)(void *arg);
};

void sel_init(struct sel_t *sel, struct sel_config *conf, int nbconf);
void sel_accept(struct sel_t *sel, char *message_name);
void sel_acceptAny(struct sel_t *sel, char **tab_message_name, int nbmsg);
void sel_call((struct sel_t *sel, char *message_name, void *arg);
```

La sémantique est la suivante :

— un **sel_call** est bloquant jusqu'à son acceptation par un **sel_accept**, ou un **sel_acceptAny** utilisant le même nom de message.

— un **sel_accept** est bloquant jusqu'à ce que la condition de garde (**fgarde**) renvoie **true** ET qu'un message de même nom soit envoyé par un **call**.

— un **sel_acceptAny** est bloquant jusqu'à ce qu'un des messages soit accepté, en respectant la garde associée au message comme pour **sel_accept**.

— les actions associées (la fonction faction) aux messages sont exécutées avant que les appels **call** et **accept** ne se terminent.

— lorsqu'un message est accepté, la fonction **faction** correspondante est exécutée. Elle reçoit en argument la valeur de l'argument **arg** du **call**.

Le code du serveur de sémaphore en C est :

```
int semserveur_compteur;
void semserveur_init(void *arg) { semserveur_compteur= (int) arg; }
void semserveur_post(void *arg) { semserveur_compteur++; }
bool semserveur_garde_wait() { return semserveur_compteur > 0; }
void semserveur_wait(void *arg) { semserveur_compteur--; }

bool FTRUE() { return true; }

struct sel_t sel;
struct sel_config selconf[] = {
      { FTRUE, "init",  semserveur_init},
      { FTRUE, "post",  semserveur_post},
      { semserveur_garde_wait, "wait",  semserveur_wait}
}

SemServeur() { // fonction executee par le thread gerant le semaphore
  sel_accept(& sel, "init");
  while(1) {
    char *tabmsg[] = {"wait", "post"};
    sel_acceptAny(&sel, tabmsg, 2);
}

Init() {
  sel_init(& sel, selconf, 3);
}

SemUtilisateur() { // exemple d'appel au thread gerant le semaphore
  sel_call(& sel, "wait"); // bloquant jusqu'a acceptation du message
  sel_call(& sel, "post"); // bloquant jusqu'a acceptation du message
}
```

Pour simplifier le code, le contenu du tableau **selconf** de type **struct sel_config** sera considéré comme stable et n'a pas besoin d'être copié.

Pour comparer deux chaînes vous pouvez utiliser la fonction **int strcmp(const char *s1, const char *s2);** qui renvoie 0 si les deux chaînes sont identiques (-1 ou $+1$ sinon en fonction de l'ordre alphabétique).

1. En utilisant les moniteurs, proposez une implantation des fonctions **sel_init**, **sel_call**, **sel_accept**, **sel_acceptAny**.
   Vous préciserez aussi le contenu de la structure **struct sel_t**.
   Vous reprendrez les notations des moniteurs utilisées en TD.
   Pour les moniteurs, vous utiliserez la sémantique des PThread (Java, etc.) pour la fonction signal, c'est-à-dire qu'elle ne donne pas la main immédiatement au processus réveillé.
   Conseils : Dans les attentes, vous devriez peut-être séparer le premier thread posant chaque message, de tous les autres threads appelant, et utiliser le **signalAll/broadcast**.

Solution:

```
struct att {
  cond_t first_caller;
  bool exist_first_caller;
  int arg;
}

struct sel_t {
  mutex_t m;
  cond_t task;
  cond_t callers;
  int nbcallers;
  struct att *att;
  struct sel_config *conf;
  int nbconf;
};

struct sel_config { bool (*fgarde)(), char *message_name, void (*faction)(void *arg

void sel_init(struct sel_t *sel, struct sel_config *conf, int nbconf) {
  lock(&sel->m);
  sel->att = malloc(nbconf * sizeof(struct att));
  sel->conf = conf;
  sel->nbconf = nbconf;
  lock(&sel->m);
};

int INSEL(sel_t *sel, char * msg) {
  int i=-1;
  for (i=0; i < sel->nbconf; i++) {
    if (! strcmp(msg, sel->conf[i].message_name))
       break;
  }
  assert(i > 0);
  return i;
}


void sel_accept(struct sel_t *sel, char *message_name) {
  lock(&sel->m);
  bool done=false;
  int n = INSEL(sel, message_name);
  while(! done) {
```

```
    while (! sel->conf[n].fgarde() || ! sel->att[n].exist_first_caller) {
      wait (&sel->task, &sel->m)
    }
    sel->conf[n].faction(sel->att[n].arg);
    signal(sel->att[n].first_caller);
  }
  unlock(&sel->m);
};

void sel_acceptAny(struct sel_t *sel, char **tab_message_name, int nbmsg) {
  lock(&sel->m);
  bool done=false;
  int n = INSEL(sel, message_name);
  while(! done) {
    bool trouve = false
    while(! trouve) {
      for(int i=0; i < sel->nbconf; i++) {
        if (sel->conf[n].fgarde() && sel->att[n].exist_first_caller) {
          trouve = true;
          break;
        }
      }
      wait (sel->task, sel->m)
    }
    sel->conf[i].faction(sel->att[i].arg);
    signal(&sel->att[i].first_caller);
  }
  unlock(&sel->m);
}

void sel_call((struct sel_t *sel, char *message_name, void *arg) {
  lock(&sel->m);
  int n = INSEL(sel, message_name);
  while(sel->att[n].exist_first_caller) {
    wait( & sel->callers, &sel->m) ;
  }
  sel->att[n].exist_first_caller = true;
  sel->attn[n].args = args;
  sel->attn[n].args = args;
  signal(&sel->task);
  wait(& sel->att[n].first_caller, &sel->m);
  sel->att[n].exist_first_caller = false;
  signalAll(& sel->callers);
  unlock(&sel->m);
}
```