Université Nice Sophia Antipolis – Polytech'Nice Sophia
Département informatique – 4ème année cycle ingénieur

Examen
Programmation Concurrente - CORRECTION

Documents autorisés : documents manuscrits + polycopié
Durée : 2h00

*Il est demandé de rédiger les solutions dans un pseudo langage (Pascal, Ada, Java…) ; lorsque **des notations et des indications sont données dans l'énoncé, il est indispensable de les respecter**, faute de quoi la solution sera considérée comme fausse.*

*La **clarté et la concision** des solutions mais aussi **l'absence de rature** seront des éléments importants d'appréciation : lorsqu'il est demandé des programmes, ceux-ci doivent être auto-suffisant. Les éventuelles explications complémentaires devront être très concises.*

*On prendra un soin particulier à la **rédaction des réponses en évitant l'a-peu-près ou l'ambiguë**. En effet, en programmation concurrente, puisqu'une solution presque juste, provoque les mêmes conséquences qu'une solution fausse, elle aura la même évaluation.*

Quelques rappels ou précisions.

Pour toutes les primitives proposées et que vous devez utiliser, on fera l'hypothèse qu'elles ne retournent pas d'erreur i.e. que les éléments demandés existent bien.

Il est possible d'utiliser des objets de type file, manipulables par les fonctions ou procédures suivantes :

```
entrer(element, file) ;  // insère un élément à la fin de la file
sortir(file) : element ; // sort le premier élément de la file
                         // si pas d'élément, retourne NULL
vide(file) : bool ;      // renvoie vrai si la file est vide
```

Voici pour rappel les primitives, inspirées de Posix, que vous pouvez utiliser pour créer un moniteur que ce soit entre thread ou entre processus :

```
typedef ... mutex_t ;
typedef ... cond_t ;
void mutex_lock(mutex_t *m);
void mutex_unlock(mutex_t *m);
void cond_wait(cond_t *c, mutex_t *m);
void cond_signal(cond_t *c);
void cond_bcast(cond_t *c);
```

Voici pour rappel les primitives, inspirées de Posix, que vous pouvez utiliser pour les sémaphores que ce soit entre thread ou entre processus :

```
typedef ... semaphore_t ;
void init(semaphore_t *s, int init); // init est la valeur
                                     // d'initialisation du sémaphore
void down(semaphore_t *s);
void up(semaphore_t *s);
```

Dans un environnement réparti, c'est-à-dire composé de plusieurs ordinateurs interconnectés qui communiquent par messages. Il existe, sur chaque machine, un système de communication qui permet à chaque processus d'envoyer un message à un autre processus et de recevoir des messages. Si l'on souhaite une réponse au message envoyé, il ne faut donc pas oublier d'inclure dans le message, son identification.

Les primitives à utiliser sont :

```
/* pour construire un message */
message := [partie 1, partie 2, partie 3]
/* pour récupérer les différentes parties d'un message */
[partie 1, partie 2, partie 3] := message
/* pour envoyer un message */
envoyer (processus, message) /* ne renvoie pas d'erreur.
                              */
/* pour recevoir un message */
message := recevoir ()      /* bloque le processus tant
                             * qu'un message n'est pas reçu
                             */
```

Dans un système centralisé, envoyer un message à un processus peu aussi consister à le déposer dans une boite aux lettres où ce processus viendra la retirer. Les boites aux lettres sont des objets de type **tampon** qui peuvent être par exemple mis en œuvre par des moniteurs munis des procédures :

```
deposer (boite, msg)  /* on fera l'hypothèse que la boite
                       * aux lettres est de taille infinie
                       */
msg := retirer (boite) /* en l'absence de message,
                       * la primitive retirer est
                       * bloquante.
                       */
```

Il est possible de construire des tableaux de boites aux lettres pour que chaque processus **p** puisse disposer d'une boite privée **BalPriv[p]** sur laquelle il peut attendre de recevoir une réponse. On dispose d'une fonction qui retourne le numéro du processus qui est en train de s'exécuter :

```
mpid := moimeme
```

Le barbier

The Sleeping Barber Problem is often attributed to Edsger Dijkstra (1965)

Barber :

- if there are people waiting for a hair cut bring them to the barber chair, and give them a haircut
- else go to sleep

Customer:

- if the waiting chairs are all full, then leave store.
- if someone is getting a haircut, then wait for the barber to free up by sitting in a chair
- if the barber is sleeping, then wake him up and get a haircut

Pour vous aider à démarrer et surtout me rendre la correction plus aisée, voici un début de modélisation en FSP :

```
const N = 3      // N fauteuils dans la salle d'attente
const M = 2      // M fauteuils de rasage
                 // correspond aussi au nombre de coiffeurs
range T = 0..M+N+2
```

Pseudo code du client

```
entre()  // appel de la fonction entre
         // modifie les variables partagées pour indiquer qu'un client est entré dans la salle

si assis() // appel de la fonction assis
alors    // je sors rasé
         // il reste donc de la place dans le salon et plusieurs cas sont possibles
         // si un fauteuil de rasage/coiffeur est disponible alors le client doit 'réveiller' le coiffeur
         // sinon le client s'assoit dans la salle d'attente
         // il attend ensuite la fin du rasage
         // la fonction retourne vrai une fois le client rasé par le coiffeur
sinon    // je sors sans avoir été rasé
         // c'est le cas si le salon est plein la fonction 'assis()' retourne faux
         // elle modélise le fait que le client ne peut pas être rasé
finsi
```

Modélisation en FSP du client

```
CLIENT     = (entre -> DANS_SALLE),          // entre dans la salle
DANS_SALLE = (assis -> RASE                  // attend ou va au rasage
          |sort_sans_rasage -> CLIENT),      // sors sans rasage
RASE       = (sort_apres_rasage -> CLIENT)   // c'est fini
               +{debut_rasage, fin_rasage}.
```

Pseudo code du coiffeur

```
Pour toujours faire
   début_rasage ()    // le coiffeur attend s'il n'y a pas de client dans la salle
   ... le barbier rase un client
   fin_rasage ()      // le barbier a fini de raser le client et le prévient
finpour
```

Modélisation FSP du coiffeur

```
COIFFEUR  = (debut_rasage -> fin_rasage -> COIFFEUR)
       +{entre, assis, sort_sans_rasage, sort_apres_rasage}.
```

Selon l'approche prise tout au long du cours, on modélise l'objet partagé entre les processus par un processus FSP qui peut être implémenté par exemple sous la forme d'un moniteur ou avec des sémaphores. Voici un début de modélisation de l'objet salle partagé entre les processus clients et les processus coiffeurs :

```
SALLE                 = SALLE[0][0][0],
SALLE[i:T][j:T][k:T] =
    // i compte le nombre de personne présent dans le salon
    // j compte le nombre de personne assise dans le salon (en salle d'attente)
    // k compte le nombre de personne au rasage

    // on peut toujours entrer
    (entre -> SALLE[i+1][j][k]

    // le client est dans le salon, il y a de la place pour la coupe
    // il s'assoie et doit 'réveiller' un coiffeur
    |when (i > 0) && (j == 0) && (k < M) assis -> SALLE[i][j][k+1]

    // le client est dans le salon, il y a de la place mais uniquement en salle
d'attente
    // le client s'assoie et 'attend' le début du rasage
    |when (i > 0) && (j < N) && (k == M) assis -> SALLE[i][j+1][k]

    //   le client est dans le salon, il n'y a pas de place, le client sort
    |when (i > N+M && j == N && k == M) sort_sans_rasage -> SALLE[i-1][j][k]

    // un client est assis sur le fauteuil, le coiffeur peut commencer le
rasage
    |when (k==1) debut_rasage -> SALLE[i][j][k]

    // le fauteuil est libre et il y a un client dans la salle d'attente, le
client change de fauteuil et le coiffeur débute le rasage
    |when (j > 0 && k < M) debut_rasage -> SALLE[i][j-1][k+1]

    // le coiffeur a terminé sa coupe et il prévient le client qui sort
    |fin_rasage -> sort_apres_rasage -> SALLE[i-1][j][k-1]).
```

Le système complet peut être construit de la manière suivante :

```
||S = ([i:0..M-1]:COIFFEUR || [i:0..M+N+1]::SALLE || [i:M..M+N+1]:CLIENT).
```

Question 1.1 Combien y a-t-il de coiffeurs (2), de salles (1) et de clients (5) dans le système S sachant que M=2 et N=3 ?

Question 1.2 proposez le code FSP de XXXXX = `(i > 0) && (j < N) && (k == M)`

Question 1.3 proposez le code FSP de YYYYY = `(i > N+M && j == N && k == M)`

Question 1.4 proposez le code FSP de ZZZZZZ = `(k==1)`

Question 1.5 proposez le code FSP de TTTTTT = `(j > 0 && k < M)`

A partir du code FSP précédent proposez une mise en œuvre de l'objet partagé 'salle' sous la forme d'un moniteur. Vous détaillerez successivement :

Question 2.1 les variables partagées du moniteur et leur valeur d'initialisation

```
int i = 0 ;
int j = 0 ;
int k = 0 ;
mutex_t m ;
cond_t coiffeur ;
cond_t client_début_rasage;
cond_t client_fin_rasage;
```

Question 2.2 le code de la procédure entre()

```
procédure entre () {
  mutex_lock(&m) ;
  i++ ;
```

```
      mutex_unlock(&m) ;
    }
```

Question 2.3 le code de la procédure assis() → booléen

```
    fonction assis() : booléen {
      mutex_lock(&m) ;
      si (YYYYYY) alors {
           // il n'y a pas de place et je sors sans être rasé
           i-- ;
           mutex_unlock(&m) ;
           renvoie faux
      }
      si  (i > 0) && (j == 0) && (k < M) alors {
           // il y a un coiffeur qui dors, je m'assois au rasage et reveille le
coiffeur
           k++ ;
           cond_signal (&coiffeur) ;
      } sinon {
           // j'attends dans la salle d'attente
           si non XXXXX alors cond_wait (&client_début_rasage, &m) ;
           j++ ;
      }
      cond_wait(&client_fin_rasage, &m) ;
      mutex_unlock(&m) ;
      renvoie vrai ;
    }
```

Question 2.4 le code de la procédure début_rasage()

```
    procédure début_rasage () {
      mutex_lock(&m) ;
      si (TTTTT) alors {
           // un client est dans la salle d'attente
           // le coiffeur peut commencer le rasage
           j-- ;
           k-- ;
           cond_signal (&client_debut_rasage) ;
      } sinon {
           // tant que personne n'est assis sur le fauteuil de rasage
           // le coiffeur dors
           si non ZZZZZ faire cond_wait (&coiffeur, &m) ;
      }
      mutex_unlock(&m) ;
    }
```

Question 2.5 le code de la procédure fin_rasage()

```
    procédure fin_rasage () {
      mutex_lock(&m) ;
      i-- ;
      k--;
      cond_signal (&client_fin_rasage) ;
      mutex_unlock(&m) ;
    }
```

Il vous reste maintenant à imaginer une solution à base de sémaphores. Vous détaillerez successivement :

Question 3.1 les variables 'communes' utilisées par les différents processus et leur valeur d'initialisation mais aussi les sémaphores utilisés ainsi que leur valeur d'initialisation.

```
    int i = 0 ;
    int j = 0 ;
    int k = 0 ;
    semaphore_t m; init(&m, 1) ;
    semaphore_t coiffeur ; init(&coiffeur , 0) ;
    cond_t client ;
```

Question 3.2 le code de la procédure entre()

```
procédure entre () {
  down(&m) ;
  i++ ;
  up(&m) ;
}
```

Question 3.3 le code de la procédure assis() → booléen

```
fonction assis() : booléen {
  down (&m) ;
  si (YYYYYY) alors {
       // il n'y a pas de place et je sors sans être rasé
       i-- ;
       up (&m) ;
       renvoie faux
  }
  si  (i > 0) && (j == 0) && (k < M) alors {
       // il y a un coiffeur qui dors, je m'assois au rasage et reveille le
coiffeur
        k++ ;
        up (&coiffeur) ;
  } sinon {
       // j'attends dans la salle d'attente
       si non XXXXX alors {
            up (&m) ;
            down (&client_début_rasage) ;
            down (&m) ;
            j++ ;
       }
  }
  down (&client_fin_rasage) ;
  up (&m) ;
  renvoie vrai ;
}
```

Question 3.4 le code de la procédure début_rasage()

```
procédure début_rasage () {
  down (&m) ;
  si (TTTTT) alors {
       // un client est dans la salle d'attente
       // le coiffeur peut commencer le rasage
       j-- ;
       k-- ;
       up (&client_debut_rasage) ;
       up (&m) ;
  } sinon {
       // tant que personne n'est assis sur le fauteuil de rasage
       // le coiffeur dors
       si non ZZZZZ faire {
            down (&m) ;
            cond_wait (&coiffeur, &m) ;
       }
  }
}
```

Question 3.5 le code de la procédure fin_rasage()

```
procédure fin_rasage () {
  down(&m) ;
  i-- ;
  k--;
  up (&client_fin_rasage) ;
  up(&m) ;
}
```

On souhaite maintenant mettre œuvre ce problème à l'aide d'un modèle client-serveur avec sur différents sites un des processus client ou coiffeur et sur un dernier site un serveur 'salle' permettant de synchroniser les processus coiffeurs et les clients. Bien évidemment dans cette approche, les clients et les coiffeurs ne communiquent pas directement entre eux mais via le serveur 'salle'. Nous proposons le code du serveur suivant :

```
Pourtoujour faire
  [émetteur, action] := recevoir()
  choix action dans
  -  entrer : entre() ; envoyer(émetteur, msg_vide)
  -  assoir : envoyer(émetteur, assis())
  -  début_rasage : envoyer(émetteur, début_rasage())
  -  fin_rasage : envoyer( émetteur, fin_rasage())
    finchoix
finfaire
```

Ce code reprend vos implémentations sous forme de moniteurs ou sémaphore des différentes procédures et on fera bien évidemment l'hypothèse que celles-ci sont correctement implémentées.

Question 4.1 est-ce que cette approche vous paraît correcte ? Si oui pourquoi et si non donnez un exemple ou cela ne fonctionne pas. 10 lignes max.

C'est correct

Question 4.2 en supposant que le code proposé ci-dessus est correct. Proposez le code d'un processus client.

```
envoyer(salle, [moimeme, entrer])
réponse := Recevoir()
si (reponse)
alors /* j'ai été rasé */
sinon /* je n'ai pas été rasé */
finsi
```

Question 4.3 en supposant que le code proposé ci-dessus est correct. Proposez le code d'un processus coiffeur.

```
envoyer(salle, [moimeme, début_rasage])
réponse := Recevoir()
/* je rase un client */
envoyer(salle, [moimeme, fin_rasage])
réponse := Recevoir()
finsi
```

Horloges scalaires ou vectorielles

Le processus A envoie aux processus B, C1 et C2 le message m1 : « Cher collègue, B va vous envoyer le document X. Pouvez-vous le traiter en urgence ? Merci », puis B envoie le message m2 au processus B, C1 et C2 : « Cher collègue, voici le document X ».

Bien évidemment pour que C1 et C2 sachent que faire du document X, il est important qu'ils aient reçu le message m1 en provenance de A avant le message m2 en provenance de B.

Question 5.1 Faut-il prendre une horloge scalaire ou une horloge vectorielle ? Comment est-elle utilisée ? Expliquer cela à l'aide d'un schéma qui mettra en évidence les messages échangés, l'évolution des horloges et les points point de délivrance du message à l'application.

Bien évidemment, il s'agit de prendre la solution la plus simple qui nécessite la manipulation de moins d'information.

On cherche a établir une relation d'ordre sur les émissions pour ordonner la délivrance des messages selon cet ordre. L'utilisation d'une horloge vectorielle qui date uniquement les émissions des messages, permet de déterminé dès la réception du message si celui-ci peut être délivré.

En supposant que les horloges sur chacun des sites démarre à 0 (horloge scalaire) ou à <0, 0, 0, 0> (horloge vectorielle) et que A < B < C1 < C2 (le 2ème élément du vecteur correspond à B) et qu'il n'y a pas d'autres échanges de message que ceux de l'énoncé.

Question 5.2 Donner le code de la procédure diffuser exécutée par un processus pour envoyer un message m à tous les processus. La procédure termine quand tous les messages ont été émis.

```
procedure diffuser (m, id, H) {
        // m est le message à envoyer
        // id est le numéro du processus (de 0 à N-1)
        // H est l'horloge du processus
        // chaque processus est identifié par un numéro
        H[id]++
        Pour tout les p de 0 à N-1 sauf id faire {
              Envoyer(id,[H,m])
        }
}
```

Question 5.3 Donne le code de la fonction délivrer exécutée par un processus pour récupérer un message. Si aucun message ne peut être délivré alors la procédure est blocante (i.e. a chaque appel elle retourne un message).

```
fonction délivrer() : message {
        délivré = faux ;

        si (non vide(file)) alors {
              [date, message] := sortir (file) ;
              si valeur(date) = valeur(H)+1 alors {
                    max_horloge (&H, &date) ;
                    délivré := true ;
                    retourner message ;
              } sinon {
                    // le message en tete de file n'est pas celui qui doit
être délivré
                    entrer (file, [date, message])
                    // la liste est ordonnée dans l'ordre croissant des dates

              }
        }
        tantque non délivré faire {
              date, message := recevoir () //
              si valeur(date) = valeur(H)+1 alors {
                    max_horloge (&H, &date) ;
                    délivré := true
                    retourner message
              } sinon {
                    // ne pas délivrer le message
                    entrer (file, [date, message])
                    // la liste est ordonnée dans l'ordre croissant des dates
              }
        }
}
```