SYSTEMES INFORMATIQUES ET APPLICATIONS CONCURRENTES
UE NFP137

DEVOIR

Le devoir comprend trois problèmes **indépendants** :

A Étude de la gestion de mémoire pour un service de messagerie noté sur 7 points
B Étude de la diffusion de messages noté sur 8 points
C À propos de lions, de moutons et d'un pont noté sur 5 points

Le devoir doit être rendu dans les séances d'ED, durant la semaine du 11 juin 2007 et au plus tard le samedi 16 juin.

A Étude de la gestion de mémoire pour un service de messagerie

Un service de messagerie est chargé de recevoir des messages envoyés par des utilisateurs et de les distribuer aux utilisateurs destinataires. À chaque utilisateur, sont associés un processus et une mémoire pour stocker les messages. Le système dispose d'une mémoire de M cases, on suppose pour simplifier que chaque case peut contenir un message et un seul. Lorsque le service de messagerie reçoit un message et une liste de destinataires, il dépose le message dans une case vide de la mémoire de chaque destinataire.
On ne se préoccupera pas dans ce problème de la synchronisation des processus, mais seulement de la gestion de la mémoire.
On fait l'hypothèse qu'on attribue initialement à chaque utilisateur un tampon de C cases pour recevoir ses messages. On réserve pour Max utilisateurs potentiels C*Max cases. Il reste M-C*Max cases de mémoire que nous appellerons "réserve" (M> C*Max).
La lecture d'un message par l'utilisateur dans son tampon n'est pas destructive, c'est-à-dire qu'elle ne libère pas de case du tampon.
Si le service de messagerie ne peut distribuer un message à un utilisateur parce que son tampon est plein, il prévient cet utilisateur. Celui-ci demande et attend l'attribution d'une case libre de la "réserve". Cette case est ajoutée à son tampon, dans une extension de son tampon initial.

Question 1.
Montrez que cette gestion peut conduire à un interblocage des utilisateurs.

Pour éviter le problème précédent, on "offre" une extension sur disque de leur tampon, aux utilisateurs : lorsqu'il n'y a plus de case libre dans la "réserve", on recherche une case "victime" qu'on recopie dans l'extension disque de l'utilisateur correspondant, et qu'on attribue au demandeur. Ainsi, les messages d'un utilisateur peuvent se trouver dans le tampon initial, dans une extension avec des cases de la "réserve", dans une extension sur disque ; les extensions sont éventuellement vides.

Dans un premier temps, on envisage une stratégie de remplacement global : la case victime est choisie dans l'ensemble des cases attribuées aux utilisateurs y compris dans leur tampon initial.

Question 2.
Quels sont à votre avis les avantages et les inconvénients de cette stratégie de remplacement global ?
Montrez que cette stratégie peut conduire à un phénomène de famine et n'est donc pas acceptable.

On envisage alors une stratégie de remplacement local : la case victime est choisie dans l'ensemble des cases attribuées au demandeur.

Question 3.
Quels sont à votre avis les avantages et les inconvénients de cette stratégie de remplacement local ?

Pour éviter les inconvénients précédents, on attribue exclusivement son tampon initial de C cases à chaque utilisateur. On envisage une stratégie de remplacement globale dans les cases de la "réserve" c'est-à-dire dans les extensions de mémoire des tampons et une stratégie de remplacement locale dans le tampon initial d'un utilisateur.

Question *4.*
Imaginer et décrire un algorithme complet de remplacement des cases.
On pourra utiliser plusieurs politiques selon les cas : l'ancienneté, une politique tenant compte du fait que si un message a été lu par son destinataire, il y a peu de chances que celui-ci le relise.

Question 5.
Quel pourra être le comportement du système, si on augmente le nombre d'utilisateurs potentiels Max, en conservant la même taille de mémoire M ? À quel phénomène, ce comportement correspond-il ?

B Étude de la diffusion de messages

On souhaite étudier un service de messagerie fonctionnant de la manière suivante :
Un utilisateur peut utiliser deux commandes envoyer_message et recevoir_message permettant respectivement d'envoyer un message et de recevoir un message. Lorsque l'utilisateur lance une commande, une tâche est associée à cette commande dans l'environnement de l'utilisateur. Celui-ci peut lancer en parallèle une commande envoyer_message et une commande recevoir_message.

La commande envoyer_message a pour effet de déposer le message dans un tampon de mémoire TE si ce tampon n'est pas plein sinon la commande est bloquée jusqu'à ce que le tampon ne soit plus plein.
Chaque utilisateur dispose d'une boîte (tampon de mémoire) pour recevoir ses messages.
La commande recevoir_message a pour effet de retirer un message de la boîte de l'utilisateur si celle-ci n'est pas vide, sinon la commande est bloquée jusqu'à ce que la boîte ne soit plus vide.
Le serveur de messagerie est une tâche chargée de diffuser les messages à leurs destinataires.
Cycliquement, le serveur effectue les opérations suivantes :

- Retirer un message M du tampon TE si TE n'est pas vide, sinon attendre ;
- Pour chaque destinataire du message M faire
  Si la boîte du destinataire n'est pas pleine alors déposer le message dans la boîte sinon le message est perdu pour le destinataire

Structures de données

On suppose donnée la constante entière : `Nb_utilisateurs` : nombre d'utilisateurs.

```
// identifiants des utilisateurs
typedef enum  {0,1,2,3,...,Nb_utilisateurs-1} Utilisateur;
typedef enum {false, true} boolean ;


typedef struct {
      Utilisateur Emetteur;
      boolean Liste_dest[Nb_utilisateurs];
// Liste_dest[i] = true si l'utilisateur i est destinataire,
// false sinon
      char Texte[256]; // chaîne de caractères
} Message;
```

Gestion de l'envoi des messages

On utilise les fonctions Deposer(Message M) et Retirer() pour gérer les messages envoyés par les utilisateurs.

```
//contexte commun (variables globales)
#define Max 50

Message TE[Max]; // Tampon d'envoi
SEM Mutex, Plein, Vide;

int Tete=0, Queue=0; // mod Max

void Deposer(Message M) {
/* appelée par les commandes envoyer_message des utilisateurs
  si le tampon TE est plein, l'appelant est bloqué jusqu'à ce
que le tampon ne soit plus plein
dépose le message dans le tampon TE */
} // end Deposer

Message Retirer() {
/* appelée par le serveur pour prélever un message du tampon
si le tampon TE est vide l'appelant est bloqué jusqu'à ce que
le tampon ne soit plus vide retire un message M et libère une
place du tampon TE   */
} // end Retirer

void initialiser_semaphore(){
// initialiser les valeurs des sémaphores
}
```

Question 1.

En utilisant les données et les sémaphores déclarés dans le contexte commun, programmer les fonctions `Deposer` *et* `Retirer` *et l'initialisation des sémaphores. On explicitera la synchronisation et la gestion à l'ancienneté du tampon TE.*

Diffusion des messages

On utilise les fonctions Diffuser (Message M) et Lire_message(Utilisateur i) pour gérer les réceptions de messages des utilisateurs.

```
// contexte commun : variables globales
#define Taille 20

typedef struct { Message Mess[Taille] ;
           } boite;

boite TR[Nb_utilisateurs];

int Nbmess[Nb_utilisateurs]= {0,0,0, ...,0};
```

```
SEM Nplein[Nb_utilisateurs];
SEM Mutex[Nb_utilisateurs];

int Tete[Nb_utilisateurs]={0,0,0, ...,0}; // mod Taille
int Queue[Nb_utilisateurs]={0,0,0, ...,0}; // mod Taille


void Diffuser(Message M) {
/* appelée par le serveur pour déposer un message M dans la
boite de chaque destinataire du message pour tout
destinataire i faire
si la boite de i n'est pas pleine alors déposer le message
dans la boite sinon le message est perdu pour i, attention dans
ce cas le serveur ne doit pas être bloqué */
}    //end Diffuser

Message Lire_message( Utilisateur i) {
/* appelée par la commande recevoir_message d'un utilisateur
i pour extraire un message de sa boite  si la boîte de i est
vide, l'appelant est bloqué jusqu'à ce que la boîte ne soit
plus vide retire un message de la boite */
}    // end Lire_message

void initialiser_semaphore(){
// initialiser les valeurs des sémaphores
}
```

Les boîtes des utilisateurs sont gérées à l'ancienneté; à chaque boîte on associe entre autres, un nombre de messages Nbmess qui permet de savoir si la boîte est pleine ou non.

Question 2.

En utilisant les variables et les sémaphores déclarés ici, programmer les fonctions `Diffuser` *et* `Lire_message`. *On explicitera la synchronisation et la gestion des boîtes.*

On suppose maintenant que le serveur ne libère une place du tampon TE que s'il a pu diffuser le message à tous les destinataires, si ce n'est pas le cas il met à jour la liste des destinataires non servis et le message reste dans le tampon TE, le serveur le traitera ultérieurement.

Question 3.

Montrer que cette politique évite la perte de messages pour les destinataires dont la boîte est pleine, mais qu'elle peut conduire à un blocage du système.
Aucune programmation n'est demandée.

C À propos de lions, de moutons et d'un pont

On suppose qu'on utilise un pont pour faire traverser un fleuve à des lions et des moutons. Comme les lions risqueraient de manger les moutons, le pont ne peut contenir pour une traversée qu'un seul type d'animal à la fois.

Pour simplifier, on fait l'hypothèse que le pont peut contenir un nombre illimité d'animaux.

On souhaite simuler ce fonctionnement en associant à chaque animal une tâche. Il y a deux types de tâches dont le comportement est le suivant :

```
TASK_CODE Lion() {

      Demande d'accès_Lion ;
      Traversée ;
      Arrivée_Lion ;

} // end Lion

TASK_CODE Mouton() {

      Demande d'accès_Mouton ;
      Traversée ;
      Arrivée_Mouton ;

} // end Mouton
```

Les règles de fonctionnement sont les suivantes :

-Si le pont est vide, alors un lion ou un mouton peut accéder au pont.

-Si le pont contient un lion (respectivement un mouton) alors les moutons (respectivement les lions) doivent attendre et les lions (respectivement les moutons) peuvent accéder au pont. Lorsque le dernier lion (respectivement le dernier mouton) a traversé, alors il libère l'accès au pont.

Question 1.

En utilisant des sémaphores et des variables d'état, écrire une solution respectant les règles de fonctionnement. Cette solution doit être sans interblocage, mais ne sera pas équitable.

Indication : on pourra se référer au problème de l'exclusion mutuelle entre deux classes de lecteurs.

Question 2.

Modifier la solution précédente pour assurer qu'au moins 4 animaux ont accès au pont avant d'effectuer la traversée **.**