SIAC Devoir corrigé indicatif

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

Question 1.

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

Si toutes les cases de la "réserve" sont attribuées, chaque participant après que son tampon initial soit rempli, peut être en attente d'une case; il y a interblocage :

- accès exclusif à une case
- allocation dynamique : les cases sont acquises une à une et accumulation : les cases ne sont restituées qu'à la fin du processus
- pas de préemption
-attente circulaire

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 risque d'attribuer des cases d'un tampon initial d'un utilisateur à un autre utilisateur. Une conséquence peut être qu'un utlisateur n'ait plus de tampon en mémoire et ne puisse plus en avoir...

Un autre problème est celui de la libération de son tampon initial lorsqu'il quitte le forum :

le tampon attribué à d'autres ne peut être libéré et il peut se faire qu'on ne puisse attribuer un tampon initial à un nouvel arrivant.

Question 3.

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

Il n'y a plus de risque qu'un utilisateur n'ait plus de tampon initial, mais il peut se produire qu'il n'utilise pas une partie de ses cases, auquel cas il y a mauvaise utilisation de la mémoire.

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.

Lorsqu'un message est lu, la case correspondante est marquée.

Recherche d'une case "victime" :

1 Recherche d'une case marquée dans la "réserve". On pourra faire cette recherche avec un critère d'ancienneté de chargement, c'est assez complexe à gérer.
2 Si 1 échoue : recherche d'une case marquée dans le tampon initial de l'utilisateur demandeur, s'il n'y en a pas recherche de la case contenant le message le plus ancien.
Remarque : cet algorithme paraît être équitable, en recherchant d'abord dans la "réserve", cela évite que celle-ci ne soit détenue par quelques utilisateurs anciens au détriment des nouveaux. On peut aussi remarquer que le tampon initial d'un utilisateur sera en général plein de messages anciens...

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 ?

En augmentant le nombre d'utilisateurs, on est amené à diminuer le nombre de cases mémoire pour chacun, on augmente le nombre de recopies sur disque provoquant une attente d'E/S plus longue. La file d'attente des requêtes au disque risque de croître indéfiniment et le système de "s'écrouler"!

B Étude de la diffusion de messages

Corrigé indicatif

Gestion de l'envoi des messages

. ***Question 1.***

En utilisant les données et les semaphores 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.*

```
// contexte commun
#define Max 50

Message TE[Max]; // Tampon d'envoi
SEM Mutex, Plein, Vide;
int Tete=0, Queue=0; // mod Max

void Deposer(Message M)
{
 P(Vide);
 P(Mutex);
 TE[Queue]=M;
```

```
 Queue = (Queue +1) % Max;
 V(Mutex);
 V(Plein);
} // end Deposer

Message Retirer()
{
 Message M;

 P(Plein);
 M = TE[Tete];
 Tete = (Tete+1)% Max;
 V(Vide);
 return M;
} // end Retirer ;

void initialiser_semaphore(){
// initialiser les valeurs des sémaphores
 E0(Mutex,1);
 E0(Vide,Max);
 E0(Plein,0);
}
```

Diffusion des messages

Question 2.
En utilisant les variables et les sémaphores déclarés dans le contexte commun, programmer les fonctions `Diffuser` *et* `Lire_message`. *On explicitera la synchronisation et la gestion des boîtes.*

```
// Contexte commun
#define Taille 20
typedef struct { Message Mess[Taille] ;
} boite;

boite TR[Nb_utilisateurs];

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

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

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


void Diffuser(Message M)
{
for (i=0;i<Nb_utilisateurs;i++) {
      if (M.Liste_dest[i]) {// i est destinataire
          P(Mutex[i]);
          if (Nbmess[i] < Taille) {
```

```
            //boite non pleine
            TR[i].Mess[Queue[i]]= M;
            Queue[i]= (Queue[i]+1)% Taille;
            Nbmess(i) = Nbmess(i) + 1;
            V(Nplein[i]) ;
          } // end if
          /* si la boite est pleine le message est perdu */
          V(Mutex[i]) ;
      } //end if
  }   // end for
} //end Diffuser


Message Lire_message(Utilisateur i)
{
Message M;

  P(Nplein[i]) ;
  M= TR[i].Mess[Tete[i]];
  Tete[i]= (Tete[i] + 1) % Taille;
  P(Mutex[i]) ;
  Nbmess[i] := Nbmess[i] – 1 ;
  V(Mutex[i]) ;
 return M;
} //end Lire_message


void initialiser_semaphore(){
// initialiser les valeurs des sémaphores
for (i=0;i<Nb_utilisateurs;i++) {
 E0(Nplein[i],0) ;
 E0(Mutex[i],1);
 }
} // end initialiser_semaphore
```

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.

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.

Un utilisateur dont la boîte est pleine, peut retirer des messages et le serveur pourra alors distribuer les messages en attente dans le tampon TE. Il peut se faire cependant, que certains utilisateurs ne prélèvent pas leurs messages et que le tampon TE soit plein avec des messages en attente qui ne pourront pas être diffusés et aucun nouveau message ne pourra être envoyé et donc reçu….

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 pour traverser un fleuve. 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.

```
// Contexte commun
int NL=0 ; // nombre de lions sur le pont
int NM=0 ; // nombre de moutons sur le pont
```

```
SEM MutexL, MutxM, Pont;

// initialisation des sémaphores
E0(MutexL,1) ; E0(MutexM,1) ; E0(Pont,1) ;

TASK_CODE Lion() {

     // Demande d'accès_Lion
     P(MutexL);
     NL =NL+1;
     if (NL==1) P(Pont) ;
     V(MutexL) ;

     Traversee() ;

     // Arrivée_Lion
     P(MutexL) ;
     NL =NL-1 ;
     if (NL==0) V(Pont) ;
     V(MutexL) ;
} // end Lion


TASK_CODE Mouton() {

     // Demande d'embarquement_Mouton ;
     P(MutexM) ;
     NM =NM+1 ;
     if (NM==1) P(Pont) ;
     V(MutexM) ;

     Traversee() ;

     // Arrivée_Mouton ;
     P(MutexM) ;
     NM =NM-1 ;
     if (NM==0) V(Pont) ;
     V(MutexM) ;
} // end Mouton
```

Question 2.

Modifier la solution précédente pour assurer qu'au moins 4 animaux de même race (lion ou mouton) sont autorisés à accéder au pont avant de traverser.

// Contexte commun ajouter :

```
SEM LA, MA ;
```

```
// initialisation des sémaphores
E0(LA,0) ; E0(MA,0) ;

TASK_CODE Lion () {
Int OK=1 ; //si OK=1 pas de blocage plus de 3 lions, si OK=0 blocage
     // Demande d'accès_Lion ;
     P(MutexL) ;
     NL =NL+1 ;
     if (NL==1) P(Pont) ;
     if (NL<4) OK=0 ; // attente de 4 Lions autorisés
     if (NL==4) {
          for (i=1;i<4;i++) V(LA) ;
     } // end if
     V(MutexL) ;
     if (OK=0) P(LA);

     Traversee() ;

     // Arrivée_Lion ;
     P(MutexL) ;
     NL =NL-1 ;
     if (NL==0) V(Pont) ;
     V(MutexL) ;
} //end Lion


TASK_CODE Mouton() {
Int OK=1 ; //si OK=1 pas de blocage plus de 3 moutons, si OK=0 blocage

     // Demande d'accès_Mouton ;
     P(MutexM) ;
     NM =NM+1 ;
     if (NM==1) P(Pont) ;
     if (NM<4) OK=0 ; // attente de 4 Moutons autorisés
     if (NM==4) {
          for (i=1;i<4;i++) V(MA) ;
     } // end if
     V(MutexM) ;
     If (OK=0) P(MA);

     Traversee();

     // Arrivée_Mouton ;
     P(MutexM) ;
     NM =NM-1 ;
     if (NM==0) V(Pont) ;
     V(MutexM) ;
} //end Mouton
```