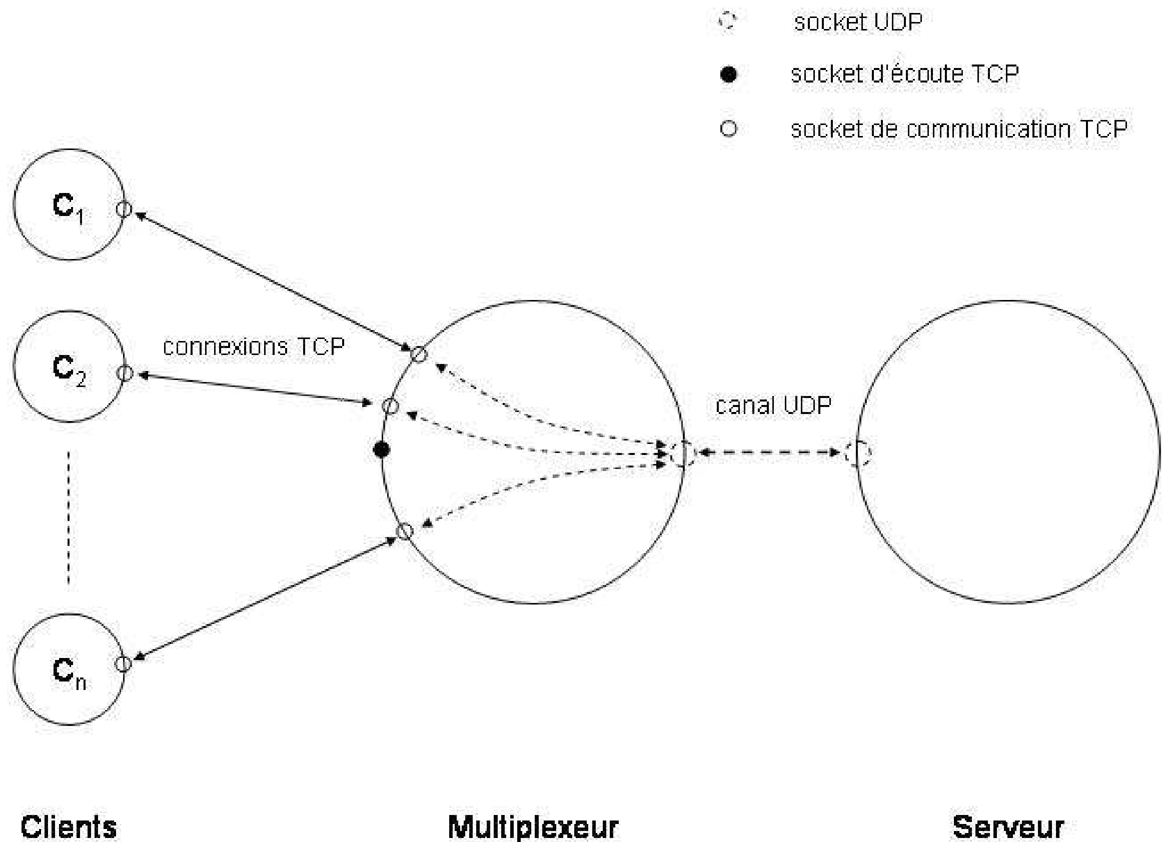
Master1 – NOYAU	- Durée : 2 heures
Contrôle Continu	- Toute documentation autorisée
Nov. 2005	- Barème indicatif
	L. Arantes, J-M. Busca et P.Sens

1. SOCKET – 8 POINTS

On souhaite développer un multiplexeur de connexions entre un serveur et ses clients. Le multiplexeur est implémenté sous la forme d'un processus unique, non multi-threadé, qui s'insère entre les clients et le serveur. Dans le DNS, on renseigne le champ « adresse IP » associé au serveur avec l'adresse IP du multiplexeur, de sorte que lorsqu'un client cherche à contacter le serveur (au travers du DNS), c'est en fait au multiplexeur qu'il s'adresse.

Chaque client établit une connexion TCP avec le multiplexeur, et le multiplexeur communique avec le serveur au travers d'un canal UDP. Le rôle du multiplexeur est d'envoyer sur le canal UDP les requêtes reçues des clients et de retourner sur les connexions TCP appropriées les réponses reçues du serveur. Dans le cadre de l'exercice, on suppose que les messages circulant sur le canal UDP ne sont ni perdus ni dé-séquencés. Le schéma ci-dessous illustre le fonctionnement du multiplexeur et ses interactions avec les clients et le serveur.



1.1 (1 point)

On suppose dans un premier temps que la connexion TCP d'un client ne peut être rompue alors que ce client est au milieu d'un échange requête – réponse avec le serveur. Expliquez comment on peut assurer le multiplexage des messages : quel est le format (en C) des messages échangés sur le canal UDP ? Quel doit être le contenu des réponses retournées par le serveur ?

Remarque : on suppose que la taille maximale d'une requête ou d'une réponse sur une connexion TCP est donnée par :

#define MAXMSG 1024

Les requêtes [resp. les réponses] circulant sur la canal UDP doivent contenir un champ identifiant le descripteur de la connexion client TCP dont elles sont issues [resp. où elles doivent être retournées]. Lorsque le serveur renvoie une réponse sur le canal UDP, il doit donner à ce champ la valeur qu'il a lu dans la requête correspondante.

```
// structure d'un message multiplexé sur le canal UDP
struct message {
  int desc;                    // descripteur de la connexion TCP d'origine
  char contenu[MAXMSG];        // contenu du message sur connexion TCP – Rmq :
};                             // seule la partie réellement utilisée du tableau est envoyée
```

Remarque : le champ desc n'est interprété que par le multiplexeur. Bien qu'il soit envoyé sur le réseau, il n'est pas nécessaire de l'écrire par htonl() et de le lire par ntohl().

1.2 (5 points)

Le squelette du code du multiplexeur est donné ci-dessous. L'argument *argv[1]* contient l'adresse du serveur et *argv[2]* contient le numéro du port du multiplexeur et du serveur (même numéro de port). Nous considérons que le serveur et le multiplexeur ne se trouvent pas sur la même machine.

Compléter les deux portions de code dont le commentaire est « initialiser le masque de descripteurs pour select » et « traiter l'événement courant ».

Pour vous simplifier l'écriture du code, on adopte les conventions suivantes :

Le premier paramètre de l'appel à select() (indice du dernier bit positionné à 1 dans le masque + 1) est systématiquement NFDBITS (constante définie dans select.h, représentant le nombre total de bits contenus dans la structure fd_set), indépendamment du nombre de bits effectivement positionnés à 1 dans le masque.

On ne traite qu'un seul événement à chaque fois que select() retourne. Si plusieurs événements sont retournés par select() et qu'un seul d'entre eux est traité, les appels suivants à select() continueront à retourner les événements non traités. En l'absence de famine, ceux-ci finiront donc par être pris en compte lors des itérations suivantes.

```
#include <stdio.h>
//...

int main(int argc, char *argv[]) {
```

```
// variables
int descUDP, descTCP;
struct hostent *serverent;
struct sockaddr_in multiaddr;
struct sockaddr_in serveraddr;

fd_set totalset, activeset;

// verifier les paramètres
if (argc != 3) {
    printf("usage: <adresse serveur> <port serveur>\n");
    exit(1);
}

// ouvrir le socket d'écoute TCP du multiplexeur
memset(&multiaddr, '\0', sizeof(multiaddr));
multiaddr.sin_family = AF_INET;
multiaddr.sin_addr.s_addr = htonl(INADDR_ANY);
multiaddr.sin_port = htons(atoi(argv[2]));

descTCP = socket(AF_INET, SOCK_STREAM, 0);
if (descTCP == -1) {
    perror("socket tcp");
    exit(1);
}
if (bind(descTCP,(struct sock_addr *)&multiaddr,sizeof(multiaddr))== -1){
    perror("bind tcp");
    exit(1);
}
if (listen(descTCP, 5) == -1) {
    perror("listen");
    exit(1);
}

// rechercher et initialiser l'adresse du serveur
serverent = gethostbyname(argv[1]);
if (serverent == NULL) {
    fprintf(stderr, "gethostbyname: host not found");
    exit(1);
}
memset(&serveraddr, '\0', sizeof(serveraddr));
serveraddr.sin_family = serverent->h_addrtype;
memcpy(&serveraddr.sin_addr, serverent->h_addr, serverent->h_length);
serveraddr.sin_port = htons(atoi(argv[2]));

// ouvrir le socket UDP de communication avec le serveur
descUDP = socket(AF_INET, SOCK_DGRAM, 0);
```

```
    if (descUDP == -1) {
        perror("socket udp");
        exit(1);
    }

    // initialiser le masque de descripteurs pour select
    // à compléter

    // boucle infinie :
    // - établissement et rupture des connexions TCP
    // - relais des messages TCP - UDP dans les deux sens
    while (1) {

        // attendre le prochain événement sur socket
        memcpy(&activeset, &totalset, sizeof(totalset));
        if (select(NFDBITS, &activeset, NULL, NULL, NULL) == -1) {
            perror("select");
            exit(1);
        }

        // traiter l'événement courant
        // à compléter
    }
}
```

Le code ci-dessous tient compte du fait que (i) si un client utilise une connexion TCP, c'est en général pour effectuer plusieurs échanges requête – réponse : le multiplexeur ne doit pas fermer la connexion après avoir renvoyé la réponse à la première requête, et laisser cette fermeture à l'initiative du client (ii) en général, le client établit une connexion dans sa phase d'initialisation, et ne l'utilise pas tout de suite pour envoyer une requête : le serveur ne doit pas faire un recv() juste après accept(), sinon il se bloque et met en attente tous les autres événements (réceptions de réponse sur le canal UDP, réceptions de requête sur les autres connexions TCP).

```
// initialiser le masque de descripteur pour select
// on mémorise dans totalset l'ensemble des descripteurs sur lesquels un événement
// peut se produire. A l'initialisation, il n'y a que le descripteur du socket UDP et le
// descripteur du socket d'écoute TCP. Plus tard, on ajoutera [resp. retirera] les des-
// cripteur de socket de communication, au fur et à mesure de l'établissement [resp.
// la rupture] des connexions client.
FD_ZERO(&totalset);
FD_SET(descTCP, &totalset);
FD_SET(descUDP, &totalset);
```

```
// traiter l'événement courant
// on rajoute les variables locale suivantes
int desc, len;
struct message buffer;
struct sockaddr_in from;
socklen_t fromlen;

// rechercher le premier descripteur de socket sur lequel il y a un événement
for (desc = 0; desc < NFDBITS && !FD_ISSET(desc, &activeset); desc++);
if (desc >= NFDBITS) {                          // par précaution (ne devrait pas arriver)
   continue;
}

// 1- cas où on est réveillé sur le socket UDP
if (desc == descUDP) {

   // le serveur a envoyé une réponse : lire cette réponse
   fromlen = sizeof(from);
   if ((len = recvfrom(desc, &buffer, sizeof(buffer), 0,
               (struct sock_addr *)&from, &fromlen) == -1) {
      perror("recv server");
      exit(1);
   }
   // si le message n'a en fait pas été émis par le serveur
   if (fromlen != sizeof(serveraddr) || memcmp(&from, &serveraddr, fromlen)) {
       // ignorer le message
       continue;
    }
   // retransmettre la réponse sur la connexion TCP cliente correspondante
   len -= sizeof(buffer.desc);
   if (send(buffer.desc, &buffer.contenu, len, 0) != len) {
      // si problème sur cette connexion, la fermer coté multiplexeur ; ne pas faire exit(),
      // les autres connexions, le socket d'écoute TCP et le socket UDP peuvent
      // continuer à fonctionner correctement
      FD_CLR(buffer.desc, &totalset);
      close(buffer.desc);
   }
```

```
    // 2- cas où on est réveillé sur le socket d'écoute TCP
    } else if (desc == descTCP) {

      // un client a fait un connect() : accepter la nouvelle connexion
      if ((desc = accept(descTCP, NULL, NULL)) == -1) {
        perror("accept");
        exit(1);
      }
      // ajouter le descripteur de la nouvelle connexion dans totalset : ainsi lors des
      // select() suivants, le multiplexeur sera prévenu des requêtes arrivant sur
      // cette connexion, ou de la rupture de cette connexion
      FD_SET(desc, &totalset);

    // 3- cas où on est réveillé sur un socket de communication TCP
    } else {

      // soit le client a envoyé une requête (recv() > 0), soit la connexion s'est
      // rompue (recv() == 0), à l'initiative du client ou sur tout autre cause
      if ((len = recv(desc, &buffer.contenu, sizeof(buffer.contenu), 0)) <= 0) {
        // en cas de problème (recv() == -1) ou de rupture de connexion (recv() == 0)
        // fermer la connexion coté multiplexeur ; ne pas faire exit()
        FD_CLR(desc, &totalset);
        close(desc);
      } else {
        // retransmettre la requête sur le socket UDP
        buffer.desc = desc;
        len +=  sizeof(buffer.desc);
        if (sendto(descUDP, &buffer, len, 0,
              (struct sockaddr *)&serveraddr, sizeof(serveraddr)) != len) {
          perror("sendto server");
          exit(1);
        }
      }
    }
```

1.3 (2 points)

On suppose maintenant qu'une connexion TCP client peut être rompue à tout moment. Le code que vous avez donné à la question 1.2 permet-il de traiter ce cas ? Si oui, expliquez comment.

Dans le cas contraire, décrivez le problème que pose la solution 1.2. Proposez ensuite un mécanisme simple permettant de le corriger : décrivez les structures de données à ajouter et/ou modifier et indiquer le principe des modifications à réaliser (on ne demande pas d'écrire le code ajouté ou modifié).

Supposons que la connexion TCP du client Ci soit rompue alors qu'une requête de Ci est en cours de traitement par le serveur. Dans la solution 1.2, le multiplexeur va détecter cette rupture et fermer le descripteur associé à cette connexion. Ce descripteur va pouvoir être réattribué à une nouvelle connexion, initiée par exemple par un autre client Cj. Lorsque que le serveur retournera sa réponse à la requête de Ci, celle-ci sera aiguillée à tord sur la connexion du client Cj.

Pour corriger ce problème, on peut identifier les connexions TCP par un numéro (pseudo-)unique attribué en séquence. Ce numéro d'identification est véhiculé dans les requêtes et réponses circulant sur le canal UDP, en plus du descripteur de connexion. Au moment de retransmettre une réponse sur une connexion TCP, on vérifie que l'identificateur de la connexion TCP courante correspond à celui du message. Si ce n'est pas le cas, la réponse n'est pas retransmise. Pour renforcer le caractère unique des identifiants, on adopte une séquence de numérotation distincte pour chaque descripteur de socket.

Structures de données :

```
// nouveau format de message multiplexé sur le canal UDP
struct message {
  int desc;
  unsigned long idcnx;                    // identifiant de connexion, relatif à desc
  char contenu[MAXMSG];
};

// table des identifiant de connexion, par descripteur
#define MAXCNX NFDBITS
static unsigned long ids[MAXCNX];

// à chaque rupture de connexion sur le descripteur desc :
ids[desc]++;

// lors de la réception d'une requête sur la connexion TCP de descripteur desc :
buffer.idcnx = ids[desc];

// lors de la réception d'une réponse sur le canal UDP
if (ids[buffer.desc] != buffer.idcnx) {
  continue;       // ignorer la réponse
}
```

Remarque : ce mécanisme n'est évidemment pas parfait. Il peut être mis en défaut si au moment où le serveur retourne une réponse, 2^32 connexions exactement ont

successivement été associées au descripteur utilisé pour véhiculer la requête. La probabilité d'un tel événement peut néanmoins être considérée comme négligeable en pratique.

2. SYNCHRONISATION – 10 POINTS

Considérez un **noyau unix non-préemptif**.

Nous voulons offrir la fonction ***int imp_fichier (int desc)*** qui permet à une application d'imprimer un fichier en passant comme argument un descripteur d'un fichier ouvert. Nous considérons qu'avant d'utiliser cette fonction l'application a ouvert le fichier en question. La fonction *int imp_fichier (int desc)* est **bloquante.** Autrement dit, le processus sera bloqué tant que l'impression du fichier n'est pas terminée (ou qu'une erreur lors de l'impression a été détectée).

Nous considérons que le périphérique d'impression offre la fonction ***lance_imp(struct inode* pinode, int *status)***, responsable de l'impression du fichier dont la référence à l'inode est passée dans l'argument *pinode* . La fin de l'impression est signalée par une interruption. A ce moment la routine ***void traite_fin_imp ()*** est exécutée. La variable passée en argument dans *status* indiquera alors si l'impression sait correctement déroulée ou non :

*status = 0 : impression correctement terminée,

* status = -1 : impression non correctement terminée.

Le système ne lance qu'une impression à la fois.

Comportement de la fonction *int imp_fichier (int desc)* :

La fonction *imp_fichier* doit premièrement sauvegarder la référence à l'*inode* du fichier à imprimer dans une case libre d'un tampon circulaire appartenant au NOYAU (tampon *imp* décrit ci-dessous). Après, **s'il n'y a pas déjà une impression en cours**, la fonction doit lancer l'impression du fichier. Pour cela, elle appelle la fonction *lance_imp* décrite ci-dessus. La fonction *imp_fichier* doit alors bloquer le processus tant que l'impression du fichier n'est pas terminée (correctement ou pas).

Code de renvoi de la fonction *int imp_fichier (int desc)* :

La fonction *imp_fichier* renvoie -2 si le tampon est plein. Sinon elle renvoie le code indiqué par la fin de l'impression.

Comportement de la routine *void traite_fin_imp ()* :

Cette routine doit faire le nécessaire pour finir la demande d'impression en cours et traiter une nouvelle demande d'impression pendante, s'il en existe dans le tampon.

Structures et variables introduites au NOYAU :

Pour implanter la fonction *int imp_fichier (int desc),* nous avons introduit au niveau NOYAU :

- une constante *MAX_IMP* qui définit la taille du tampon circulaire.
- La structure *imp_info* qui contient le pointeur vers l'inode à imprimer et le code de renvoi de l'impression en question.

```
struct imp_info {
    struct* inode  ip ;
    int code_ret ;
  }
```

- La variable global *imp* du NOYAU qui définit le tampon circulaire géré de façon FIFO.

```
struct {
  struct imp_info  buff[MAX_IMP] ; /* tampon circulaire */
  int in; /* prochaine entrée libre du tampon */
  int out ; /* prochain fichier à imprimer; premier entrée occupée*/
  int cont; /* nombre de fichier dans le tampons */
}imp;
```

- La priorité *PIMP*, définie comme la priorité de réveil pour un processus qui attend la fin d'une impression.

Observations :

- Pour le masquage/démasquage d'interruptions niveau impression vous devez utiliser les fonctions *int splimp ()* et *splx (int s)* respectivement.
- Les variables *imp.in*, *imp.out* et *imp.count* sont initialisées à 0 au démarrage du système.
- Si nécessaire, vous pouvez ajouter d'autres variables et/ou structures.

2.1 (2 points)

Est-t-il nécessaire de protéger l'accès aux variables *imp.in*, *imp.out*, *imp.count* ainsi qu'aux cases de *imp.buff* ? Justifiez votre réponse pour chacune des variables. Si c'est nécessaire, indiquez comment les protéger.

- *.imp.in* et *imp.out* : **Pas nécessaire**. On se trouve dans un noyau non préemptif. Donc la variable *imp.in* ne sera pas partagée par plusieurs « producteurs ». La variable *imp.out* ne sera accédée que par la routine d'interruption.

imp.count **: Nécessaire** : variable partagée par la fonction *imp_fichier* et *traitement_fin_imp*. Il faut la protéger. Masquage/démasquage d'interruptions splimp () et splx ().

imp.buff : **Pas nécessaire :** accès à chaque case est indépendante : *imp_fichier* utilise *imp.in* et *traitement_fin_imp* utilise *imp.out* .

2.2 (1 point)

Quel problème se poserait si nous choisissions l'adresse de l'inode du fichier pour endormir le processus en attente de la fin l'impression du fichier en question (paramètre de la fonction *sleep*)? Quelle adresse pensez vous choisir ?

Il se peut que plusieurs processus aient demandé d'imprimer le même fichier. La fonction d'interruption réveillerait alors tous les processus en attente sur cette adresse et non pas forcement celui qui a fait la demande.

Suggestion d'adresse : case du tampon où se trouve la demande d'impression.

2.3 (1 point)

A votre avis, la priorité PIMP (et en conséquence la fonction *imp_fichier*) peut-elle être interruptible aux signaux ? Justifiez votre réponse.

[Plusieurs réponses ont été acceptées, si elles étaient correctement argumentées]

Une impression pouvant durer longtemps (voire ne pas se terminer si l'imprimante est en panne), il est utile de pouvoir interrompre un processus en attente de fin d'impression (par exemple avec Ctrl-C), de façon à le terminer ou à lui permettre d'effectuer d'autres traitements.

Cependant, dans le cas où le processus termine son exécution après réception du signal, il est possible que l'inode du fichier à imprimer soit libéré, alors que sa référence restera présente dans la file imp.buff : lance_imp() accèdera à un inode soit incorrect soit différent de celui du fichier à imprimer. Pour corriger ce problème, on peut incrémenter le nombre de références à l'inode du fichier à imprimer dans lance_imp() et le décrémenter dans traite_fin_imp().

Hormis ce problème, rendre lance_imp() interruptible par les signaux ne crée pas d'incohérence dans la gestion des champs de la structure imp. Il faut simplement modifier la solution donnée à la question 2.4 pour retourner -1 avec la variable errno positionnée à EINTR dans le cas d'une interruption par un signal. Il faut aussi préciser dans la spécification de imp_fichier() que même si l'appel est interrompu, l'impression a quand même été lancée : le processus ne doit donc pas reboucler sur imp_fichier(), sinon une seconde impression du même fichier sera lancée.

2.4 (5 points)

Donnez le code de la fonction ***int imp_fichier (int desc).*** Vous devez faire de sorte qu'elle soit le moins possible non interruptible (sections critiques les plus courtes possibles). Donnez aussi le code de la routine ***void traite_fin_imp ()***.

```
int imp_fichier (int desc) {
 struct inode * ip = u.u_ofile[desc] ->f_inode;
  int s , int I;

  s= splimp();  /* Formellement, il n'est pas nécessaire de protéger cette comparaison. */
   if (imp.cont == MAX_IMP) {
     splx(s);       /* La protection est en revanche nécessaire si, suite au test, on s'endort     */
    return (-2);  /* en attendant qu'une entrée du buffer se libère. Dans ce cas, on ne doit */
  }                     /* pas se démasquer avant d'appeler sleep() : c'est sleep() qui le fera.       */
  splx(s);
```

```
  I = imp.in ;
 /* inclure l'inode dans le buffer */
  imp.buff[I].ip = ip;
  /* indiquer que l'impression n'est pas terminée */
  imp.buff[I].code_ret = IMP_PAS_FINIE ;

 /* incrémenter le pointeur de case libre */
  imp.in = (imp.in+1) %MAX_IMP ;

 /* incrémenter le compteur */
  s= splimp();
  imp.cont++ ;
   if (imp.cont == 1)
        lance_imp (ip, &imp.buff[I].code_ret);
   /* les interruptions imprimantes étant masquées, l'interruption signalant la fin de l'im-
       pression que l'on vient de lancer n'a pas pu être traitée : on peut s'endormir sans tester
      si l'impression est terminée ou non. */
sleep (&(imp.buff[I]),PIMP) ;
  splx (s);
  return (imp.buff[I].code_ret);
}

int traite_fin_imp () {
 wakeup(&imp.buff[imp.out]);
 imp.out = (imp.out +1) % MAX_IMP ;
 imp.count --;
 if (imp.count >0)
    lance_imp (imp.buff[imp.out].ip,imp. &buff[imp.out].code_ret);
}
```

2.5 (1 point)

Si le noyau était préemptif, est-ce que votre réponse à la question 2.1 changerait ? Justifiez votre réponse.

La variable *imp.in* serait partagée par plusieurs processus. Alors l'accès à cette variable et le remplissage de la case du buffer indexée par elle doivent être protégés.

imp.out : reste le même. Pas de protection nécessaire.

imp.cout : reste le même. Protégé.

3. Questions de Cours – 2 points

3.1 (1 point)

Quelle est la différence entre un noyau préemptif et un noyau non préemptif ?

Dans un noyau préemptif, un processus (thread) peut être interrompu (préempté) par l'ordonnanceur si un processus/ thread plus prioritaire se trouve dans l'état prêt.

Dans un noyau non préemptif, ce changement de contexte n'aura lieu que lorsque le système passe de mode noyau au mode usager.

3.2 (1 point)

Indiquez brièvement pourquoi lorsque l'on traite un signal cela indique que celui ci a été envoyé "au moins" une fois.

L'appel kill se contente de positionner dans p_sig le bit correspond au signal. Le traitement n'aura lieu que lorsque le processus sera de nouveau élu. Il est possible qu'avant d'être élu le bit d'un signal soit positionné à 1 plusieurs fois.