CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Samedi 16 juin 2007
2 heures
08h00 – 10h00

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103

CORRECTION DE L'EXAMEN
Année 2006 – 2007, second semestre

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Centres Régionaux du PTI Nord-Est

Question n°1 : répertoire téléphonique *(7 pts)*

Question 1.a) ***[1 point]*** Dans un système multiprogrammé, il faut absolument qu'un processus ait l'assurance d'être le seul à accéder à un fichier lors d'une écriture. Sinon, un autre processus lisant en même temps le fichier pourrait obtenir des données inconsistantes. Ou deux processus qui écriraient en même temps pourraient interférer l'un l'autre. Par contre, rien n'empêche à plusieurs processus de lire en même temps.

Question 1.b) ***[2,5 points]*** Trois solutions attendues pour le maximum de points :

- une solution non centralisée avec les sémaphores (algorithme des lecteurs rédacteur). Avantage de cette solution : code efficace. Inconvénient : si plusieurs programmeurs développent diverses composantes de l'applications, tous doivent respecter les mêmes règles (obligation d'utiliser les sémaphores comme convenu) ;
- une solution centralisée sous forme de moniteur : les ordres de lectures ou d'écriture sont envoyés à un processus unique, via une « API » proposée par le moniteur. Les mécanismes choisis afin de communiquer entre les processus (tubes, files de messages, etc.) assurent leur synchronisation ;
- une solution client-serveur utilisant les couches réseau. Ressemble au modèle du moniteur, excepté que clients et serveurs peuvent être potentiellement sur deux machines distinctes. La solution ne semble pas adaptée ici, les appels systèmes aux fonctions réseau seraient coûteux, alors que les processus sont sur une même machine.

Question 1.c) ***[3,5 points]*** La solution à cet exercice correspond à la résolution du problème lecteurs-rédacteur. On utilise une variable globale `nb_lect` qui contient le nombre de lecteurs à un instant T. Il faut ensuite utiliser deux sémaphores :

- un sémaphore `sem_nb_lect` qui protège l'accès à la variable globale `nb_lect`,
- et un sémaphore `sem_exclu` qui permet à un rédacteur ou à l'ensemble de tous les lecteurs de s'assurer l'exclusivité de l'accès au fichier.

Ces éléments sont initialisés au début du programme avec le code suivant :

```
int   nb_lect = 0 ;
Init(sem_nb_lect, 1);
Init(sem_exclu, 1);
```

Les fonctions « `ajouter_au_carnet_adresses()` » et « `effacer_une_entree()` » sont des fonctions ayant toutes deux un rôle de rédacteur. Leur squelette est identique :

```
int ajouter_au_carnet_adresses(char *numero, char *nom, char *prenom)
/* ou int effacer_une_entree (char *numero, char *nom, char *prenom) */
{
      int   code_retour;
      P(sem_exclu);
      /* portion de code d'accès au fichier en écriture */
      V(sem_exclu);
      return(code_retour);
}
```

Les fonctions « `quel_est_le_nom_d_expediteur()` » et « `quel_est_le_numero()` » ont exactement le même squelette, à savoir celui d'un lecteur :

```
int quel_est_le_nom_d_expediteur(char *numero)
/* ou int quel_est_le_numero(char *numero) */
{
      int   code_retour;
      /* je me garantis l'exclusivité de la variable nb_lect : */
      P(sem_nb_lect);
      nb_lect++;
      if (nb_lect == 1) /* je suis le premier lecteur */
      {
            /* je donne l'exclusion au groupe des lecteurs */
            P(sem_exclu);
      }
```

```
        /* je n'ai plus besoin d'accéder à nb_lect : */
        V(sem_nb_lect);

        /* ici, le coeur des 2 fonctions quel_est_le_nom_d_expediteur() */
        /* ou quel_est_le_numero()... bla bla bla ...  */

        /* je me garantis l'exclusivité de la variable nb_lect : */
        P(sem_nb_lect);
        nb_lect--;
        if (nb_lect == 0) /* je suis le dernier lecteur */
        {
              /* je libère l'exclusion au groupe des lecteurs */
              V(sem_exclu);
        }
        /* je n'ai plus besoin d'accéder à nb_lect : */
        V(sem_nb_lect);
        return(code_retour);
}
```

Question n°2 : mémoire, segmentation, pagination *(6 pts)*

Question 2.a) ***[1 point]*** L'utilisation de mécanismes de translation d'adresses et d'espace d'adressage virtuel offre les avantages suivants :

- possibilité d'offrir aux processus plus de mémoire qu'il n'existe de mémoire vive installée dans la machine (via le mécanisme de mémoire virtuelle/swap),
- possibilité d'isoler la mémoire allouée à un processus afin que d'autres processus ne puissent y accéder,
- possibilité de marquer certaines zones de mémoire avec des droits pour le processus en lecture, écriture, ou exécution de code. Lorsqu'elle est utilisée, cette technique permet d'éviter à un utilisateur d'exécuter du code (intentionnellement ou par accident) qu'il aurait introduit dans des zones réservées pour stocker des données [et permet la mise en œuvre d'un mécanisme de *copy on write*].

Question 2.b) ***[2 points]*** Les points communs entre la segmentation et la pagination sont :

- dans les deux concepts, la mémoire est découpée en « morceaux »,
- le CPU utilise une table pour traduire une partie de l'adresse virtuelle en adresse réelle,
- dans un processus, deux segments différents peuvent recouvrir deux portions identiques de mémoire vive (alors que deux pages pointeront vers deux cadres distincts).

Les différences sont :

- les pages sont de tailles fixes, les segments peuvent être de tailles différentes,
- en cas de pagination, dans une adresse virtuelle, le nombre de bits utilisés pour coder le décalage dans une page est choisi en fonction de la taille d'une page. Ainsi, les bits de poids faibles dans l'adresse virtuelle correspondent aux bits de poids faible dans l'adresse réelle,
- pour cette même raison, le décalage dans une adresse paginée ne peut permettre de sortir de la case correspondant à la page. Alors que le décalage dans une adresse linéaire peut permettre à l'adresse réelle de sortir de l'espace de mémoire vive correspondant au segment, levant alors une exception.

Question 2.c) ***[3 points]*** L'algorithme LRU (Least Recently Used/la moins récemment utilisée) consiste à choisir comme victime la page qui n'a pas été référencée depuis le plus longtemps.

Page demandée	2	1	0	7	1	6	1	5	7	6	2	6	7	0	7	1	0	2	1	7
Case 0	2	2	**2**	7	7	7	**7**	5	5	**5**	2	2	**2**	0	0	0	0	0	**0**	7
Case 1		1	1	1	1	1	1	1	**1**	6	6	6	6	6	**6**	1	1	1	1	1
Case 2			0	0	**0**	6	6	**6**	7	7	7	7	7	7	7	7	**7**	2	2	2
Défaut de page				O	N	O	N	O	O	O	O	N	N	O	N	O	N	O	N	O

A la fin de tous les accès, la table des pages est alors :

page	bit V	case
0	0	/
1	1	1
2	1	2
3	0	/
4	0	/
5	0	/
6	0	/
7	1	0

Remarque : noter juste ce tableau même si le tableau précédent est faux, du moment ou le candidat sait retrouver cette table des pages à partir de la table des cases à laquelle il a abouti.

Question n°3 : stockage RAID1 réseau *(7 pts)*

Question 3.a) ***[1 point]*** Même si la fonction `pipe()` crée deux descripteurs, un tube est toujours monodirectionnel. Une fois qu'il est utilisé dans un sens, il n'est plus possible de l'utiliser dans l'autre sens (un des deux descripteurs créé devient inutile pour chacun des deux processus). Il faut ainsi deux tubes pour communiquer dans les deux sens entre le père et le fils n°0, et deux tubes pour la communication père/fils n°1.

Question 3.b) ***[2 points]***

```
while (1)
{
      /* On attend une requête */
      taille_msg_recu=recvfrom(mon_socket, &question, sizeof(question), \
           0, (struct sockaddr *)&sockadrs_client, &lg_sockadrs_client);
      if (taille_msg_recu < 0)
      {
            fprintf(stderr,"Erreur à la réception d'une trame\n");
            exit(-1);
      }
      memcpy(&reponse,&question,TAILLE_BLOC);
      /* On prépare la réponse */
      switch (ntohs(question.operation))
      {
            case  OP_LECTURE :
                  err=lit_bloc(ntohl(question.num_bloc), reponse.bloc);
                  break;
            case  OP_ECRITURE :
                  err=ecrit_bloc(ntohl(question.num_bloc), question.bloc);
                  break;
            default     :
                  fprintf(stderr, "Erreur fatale: question inconnue\n");
                  exit(-1);
      }
      reponse.operation=htons((err==0)?OP_OK:OP_ERREUR);
      /* Et on envoie la réponse */
      if (sendto(mon_socket, &reponse, sizeof(reponse), 0, \
          (struct sockaddr *)&sockadrs_client, lg_sockadrs_client ) <  0)
      {
            fprintf(stderr,"Erreur à l'émission d'une trame\n");
            exit(-1);
      }
}
```

Question 3.c) ***[4 points]***

- « `fin_demandee()` » est exécutée par les fils lorsque le père envoie aux fils un signal `SIGUSR1`. Cette fonction (et par conséquent, l'envoi par le père d'un signal `SIGUSR1`) vise à faire mourir le fils. Dès lors, le fils passe à l'état de zombie. La boucle « `while (wait(NULL) > 0) ;` » exécutée par le père permet de détruire ces zombies, libérant ainsi les ressources que ces derniers conservaient dans les tables et structures système gérant les processus.
- « nb_fils » est une variable globale. Chaque processus ayant sont propre espace d'adressage, cette variable peut avoir une valeur différente pour chacun des processus. Au tout début de l'exécution du programme, il n'existe qu'un processus (le père). Lors de l'arrivée dans la boucle « `for (nb_fils = 0 ; nb_fils < 2 ; nb_fils++) { le_pid = fork(); ...` », la variable `nb_fils` est initialisée à 0. Puis :
  - au premier appel `fork()`, le premier fils créé (le fils n°0) hérite d'une variable `nb_fils` qui vaut 0. Elle n'est plus modifiée ensuite ;
  - au deuxième passage dans la boucle, le deuxième appel `fork()` crée le second fils (le fils n°1), qui hérite d'une variable `nb_fils` qui vaut 1 ;

  Le père sort de la boucle à cause de la condition « `nb_fils < 2` » qui devient fausse, lorsque `nb_fils` vaut 2. Cette variable n'est plus modifiée ensuite. Pour résumer, `nb_fils` vaut :
  - 0 pour le premier fils,
  - 1 pour le second fils,
  - et successivement 0, 1, et 2 pour le père. Elle reste à 2 une fois les fils créés.
-

```
while(1)
{
      /* Attente d'un ordre de la part du père sur le pipe */
      if (read(tubes_pere_vers_fils[nb_fils][0], &msg, sizeof(msg)) < 0)
      {
            fprintf(stderr,"Erreur du fils à lire dans un pipe\n");
            exit(-1);
      }

      /* On standardise la question et le numéro de bloc pour le réseau */
      msg.operation=htons(msg.operation);
      msg.num_bloc=htonl(msg.num_bloc);

      /* On envoie la requête */
      if (sendto(mon_socket, &msg, sizeof(msg), 0, \
       (struct sockaddr *)&sockadrs_serv, sizeof(struct sockaddr_in)) < 0)
      {
            fprintf(stderr,"Erreur à l'émission d'une trame\n");
            exit(-1);
      }

      /* On attend la réponse */
      FD_ZERO(&read_fs);
      FD_SET(mon_socket, &read_fs);
      if (select(mon_socket+1, &read_fs, NULL, NULL, &t_out) <= 0)
      {
            /* Time out : pas de réponse au bout de 3 secondes... */
            msg.operation=OP_TIMEOUT;
      }
      else
      {
            /* Pas de timeout, il y a une réponse à lire */
            taille_msg_recu=recvfrom(mon_socket, &msg, sizeof(msg), \
                                     0, NULL, 0);
```

```
                if (taille_msg_recu < 0)
                {
                      fprintf(stderr,"Erreur à la réception d'une trame\n");
                      exit(-1);
                }
                /* On standardise la réponse et le numéro de bloc */
                msg.operation=ntohs(msg.operation);
          }
          msg.num_bloc=ntohl(msg.num_bloc);

          /* On envoie la réponse au père par le pipe */
          if (write(tubes_fils_vers_pere[nb_fils][1], &msg, sizeof(msg)) < 0)
          {
                fprintf(stderr,"Erreur à l'écriture dans le pipe\n");
                exit(-1);
          }
     }
```