CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Lundi 16/06/2008
2 heures

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103

CORRECTION DE L'EXAMEN
Année 2007 – 2008, deuxième semestre
Coefficient : 2/3

Documents autorisés : NON

Cnam
CONSERVATOIRE NATIONAL
DES ARTS ET METIERS

Exercice n°1 : simulateur de vol *(4 questions pour un total de 8 pts)*

Question 1.a) [*1 point*] :

```
void detection_collision()
{
    affiche_explosion();
    if (change_nb_vies(-1) == 0)
        game_over();
    else
    {
        /* Il faut penser à réarmer la fonction detection_collision() */
        /* à la réception de SIG_USR1. C'est cette ligne qui donne le point. */
        signal(SIG_USR1, detection_collision);
    }
}
```

Question 1.b) [*2 points*] :

Premier scénario : le processus fils détecte la collision avec l'immeuble, et envoie le signal `SIG_USR1` à son père. Puis, hasard du programme, il y a préemption. Le CPU est donné au programme principal, qui a reçu le signal `SIG_USR1`. Dans ce cas, la fonction `detection_collision()` est exécutée, et le joueur perd une vie. Soit il y a un « game over », soit le joueur retourne au début de son action (il ne percutera pas le pilonne).

Second scénario : le processus fils détecte la collision avec l'immeuble (envoi du signal `SIG_USR1` au père), mais comme l'action se déroule vite, il n'y a pas de préemption avant que l'avion de rencontre le pilonne (envoi à nouveau du signal `SIG_USR1`). Or, le signal `SIG_USR1` est un **signal classique**. Aussi, le processus père ne saura pas qu'il a reçu deux fois ce signal (les signaux classiques ne s'empilent pas). La fonction `detection_collision()` ne sera donc appelée qu'une seule fois.

Dans tous les cas, le joueur ne perdra qu'une seule vie.

Question 1.c) [*1 point*] :

S'il peut y avoir plusieurs processus qui viennent lire `nb_vies`, un processus qui souhaite modifier cette variable doit s'assurer de l'exclusivité de son accès. C'est le problème dit du lecteur/rédacteur, qui se résout avec deux sémaphores et une variable globale indiquant le nombre de lecteurs.

Question 1.d) [*4 points*] :

```
/* variables globale : */
int nb_vies;
int nb_lect = 0;
int ensemble_sem;
#define  MA_CLE 0x00001234
struct sembuf operations[2];

/* code pour initialiser les 2 sémaphores */
/* remarque : le sémaphore 0 permet d'assurer l'exlusivité sur */
/* la variable nb_vies, et le sémaphore 1 permet d'assurer l'exlusivité */
/* sur la variable nb_lect */
ensemble_sem = semget(MA_CLE, 2, IPC_CREAT|S_IRUSR|S_IWUSR);
```

```
/* les deux sémaphores sont initialisés à 1 */
semctl(ensemble_sem, 0, SETVAL, 1);
semctl(ensemble_sem, 1, SETVAL, 1);


/* bla bla bla */


int change_nb_vies(int v)
{
    int ret;
    /* P(semaphore N°0) */
    operations[0].sem_num = 0;
    operations[0].sem_op = -1;
    operations[0].sem_flg = 0;
    semop(ensemble_sem, operations, 1);
    nb_vies = nb_vies + v;
    ret = nb_vies;
    /* V(semaphore N°0) */
    operations[0].sem_num = 0;
    operations[0].sem_op = 1;
    operations[0].sem_flg = 0;
    semop(ensemble_sem, operations, 1);
    return(ret);
}


int lit_nb_vies()
{
    int ret;
    /* P(semaphore N°1) */
    operations[0].sem_num = 1;
    operations[0].sem_op = -1;
    operations[0].sem_flg = 0;
    semop(ensemble_sem, operations, 1);
    nb_lect++;
    if (nb_lect == 1) /* je suis le premier lecteur */
    {
        /* je donne l'exclusion au groupe des lecteurs */
        /* P(semaphore N°0) */
        operations[0].sem_num = 0;
        operations[0].sem_op = -1;
        operations[0].sem_flg = 0;
        semop(ensemble_sem, operations, 1);
    }
    /* V(semaphore N°1) */
    operations[0].sem_num = 1;
    operations[0].sem_op = 1;
    operations[0].sem_flg = 0;
    semop(ensemble_sem, operations, 1);
    ret = nb_vies; /* lecture de la varaible critique */
    /* P(semaphore N°1) */
    operations[0].sem_num = 1;
    operations[0].sem_op = -1;
    operations[0].sem_flg = 0;
    semop(ensemble_sem, operations, 1);
    nb_lect--;
    if (nb_lect == 0) /* je suis le dernier lecteur */
    {
        /* je libère l'exclusion au groupe des lecteurs */
        /* V(semaphore N°0) */
        operations[0].sem_num = 0;
        operations[0].sem_op = 1;
```

```
        operations[0].sem_flg = 0;
        semop(ensemble_sem, operations, 1);
    }
    /* V(semaphore N°1) */
    operations[0].sem_num = 1;
    operations[0].sem_op = 1;
    operations[0].sem_flg = 0;
    semop(ensemble_sem, operations, 1);
    return(ret);
}
```

Remarque du correcteur : une solution avec deux ensembles de un seul sémaphore sera acceptée aussi. Une solution avec les classiques fonctions Init(), P(), et V(), sans détail des appels système Linux ne comptera que pour la moitié des points.

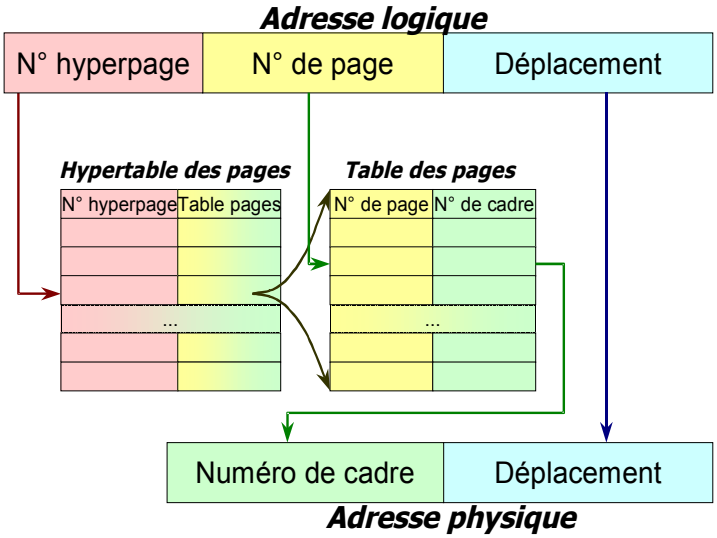
Exercice n°2 : mémoire, segmentation, pagination *(3 questions pour un total de 6 pts)*

Question 2.a) [*1 point*] : La pagination.

Question 2.b) [*2 points*] : La MMU ne pouvant charger toute la table des pages en mémoire, les fondeurs ont mis en place la **pagination multiniveaux** (ou **hyperpagination**), avec la notion d'hypertable. L'adresse virtuelle ne contient plus deux sous ensembles (la page et le décalage), mais 3 : l'hyperpage, la page, et le décalage. L'accès à la mémoire physique se fait alors avec un accès supplémentaire :

- la lecture de l'hyperpage est utilisée comme entrée dans l'hypertable, et nous permet d'accéder à une table des pages ;
- la page permet de trouver le cadre dans la table des pages ainsi obtenue ;
- la lecture de la donnée dans la mémoire physique se fait à l'aide de ce cadre et du décalage.

Schéma de la traduction d'une telle adresse :



Traduction d'adresse (hyperpagination)

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	5	6	7	0	6	1	6	2	0	1	5	1	0	7	0	6	7	5	6	0
Case 0	5	5	**5**	0	0	0	**0**	2	2	**2**	5	5	**5**	7	7	7	7	7	**7**	0
Case 1		6	6	6	6	6	6	6	**6**	1	1	1	1	1	**1**	6	6	6	6	6
Case 2			7	7	**7**	1	1	**1**	0	0	0	0	0	0	0	0	**0**	5	5	5
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	1	0
1	0	/
2	0	/
3	0	/
4	0	/
5	1	2
6	1	1
7	0	/

Remarque du correcteur : 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.

Exercice n°3 : IPC/MSQ *(4 questions pour un total de 6 pts)*

Question 3.a) [*1 point*] : Un MSQ permet d'envoyer des messages typés de quelques Ko entre deux processus d'une même machine. Sous Linux, ce mécanisme est réalisé à l'aide des IPC.

Question 3.b) [*1 point*] : Les MSQ ne fonctionnent qu'entre processus situés sur une même machine. Pas de problème si ces processus sont tournent sur 2 CPU différentes, mais ils ne peuvent être sur deux machines différentes.

Question 3.c) [*2 points*] : « *programmeB.c* » :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include <errno.h>

#include "mon_msgbuf.h"

void retrait(float f)
{
      /* code qui traite un retrait de f € */
      bla bla bla...
}

void depot(float f)
{
      /* code qui traite un dépôt de f € */
      bla bla bla...
}
```

```
int main()
{
      int                  code_err_main;
      int                  msqid;
      struct mon_msgbuf    msg;
      /* Création du MSQ */
      msqid = msgget(cle, IPC_CREAT|S_IRUSR|S_IWUSR);
      if (msqid < 0)
      {
            perror("Erreur à la création du MSQ\n");
            exit(-1);
      }
      /* Début correction */
      do
      {
            if (msgrcv(msqid, (void *)&msg, sizeof(msg.valeur), 0, 0) < 0)
            {
                  perror("Erreur lors d'un msgrcv(). Fin anormale.\n");
                  exit(-1);
            }
            switch (msg.mtype)
            {
                  case 1 :
                        /* cas du retrait */
                        retrait(msg.valeur);
                        break;
                  case 2 :
                        /* cas du dépos */
                        depot(msg.valeur);
                        break;
                  case 3 :
                        /* fin du programme */
                        code_err_main = (int)msg.valeur;
            }
      } while (msg.mtype < 3);
      /* On détruit proprement le MSQ */
      msgctl(msqid, IPC_RMID, NULL);
      if (code_err_main == 0)
            printf("Fin normale du programme demandée.\n");
      else
            printf("Fin anormale du programme.\n");
      return(code_err_main);
}
```

Question 3.d) [*2 points*] : Le programme principal :

- lance un premier fils. Ce fils fera un **`msgrcv(msqid, (void *)&msg, sizeof(msg.valeur), 1, 0)`** pour ne récupérer (et traiter) que les messages de type 1 (retrait) ;
- lance un second fils. Celui-ci fera un **`msgrcv(msqid, (void *)&msg, sizeof(msg.valeur), 2, 0)`** pour ne récupérer (et traiter) que les messages de type 2 (dépôts) ;
- enfin, il lancera un **`msgrcv(msqid, (void *)&msg, sizeof(msg.valeur), 3, 0)`**. A la réception d'un tel message, le père enverra un signal pour mettre fin à ses deux fils. Il aura pris soin au préalable d'armer une fonction comme handler du signal `SIG_CHLD`, afin d'aller récupérer le code retour de ses fils, évitant ainsi la création de zombies.