CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Durée : 2 heures

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103 - NANCY/METZ

INTERVENANT : E. DESVIGNE

CORRECTION DE LA SESSION DE RATTRAPAGE – 13/09/2011

Année 2010 – 2011, deuxième semestre
Coefficient : 1/1 (compte pour la totalité de la note)

Documents autorisés : NON

CONSERVATOIRE NATIONAL
DES ARTS ET METIERS

Exercice n°1 : jeu de la patate chaude *(3 questions pour un total de 8 points)*

Question 1.a) [*1 point*] : dans un environnement Unix/Linux, un « processus » est un programme dont l'exécution se déroule en partageant le (ou les) CPU avec d'autres processus. Les processus sont « isolés » les uns des autres (chaque processus a sa propre zone mémoire, ses propres descripteurs de fichiers, etc.).

Question 1.b) [*1 point*] : un pipe anonyme est un canal de communication FIFO qui permet la connunication d'un flux de données entre un père et son fils (ou vice et vers ça). Le pipe est créé par le père, et les descripteurs de fichiers sont hérités par le fils. Les pipes nommés fonctionnent suivant le même principe, mais peuvent être utilisés par deux processus sans lien de parenté direct. Les descripteurs de fichiers font alors référence à un nom de fichier (de type "pipe").

Question 1.c) [*6 points*] :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

int   tab_pid[4]; // contiendra le PID de chaque joueur
int   p[4][2]; // les 4 pipes anonymes
int   score; // le score du processus
int   nb_joueur;

int num_joueur_aleat()
{
      long int li;
      li=lrand48()%4;
      return((int)li);
}

void reception_patate_fils()
{
      int   prochain_joueur;
      signal(SIGUSR1, reception_patate_fils); // on réarme
      score++;
      if (score < 1000)
      {
            // on retire un nombre aleatoire tant qu'on se choisit nous même
            do {
                  prochain_joueur=num_joueur_aleat();
            } while (prochain_joueur==nb_joueur);
            kill(tab_pid[prochain_joueur], SIGUSR1);
      }
      else
      {
            // fin de partie, on envoie la patate au père
            kill(getppid(), SIGUSR1);
      }
}

void reception_patate_pere()
{
      // quand le père reçoit la patate, il envoie le signal SIGUSR2 aux 4
      // fils, puis il récupère leur valeur de fin pour éviter les zombies,
      // puis s'en va...
      int   i;
      for (i=0 ; i<4 ; i++)
            kill(tab_pid[i], SIGUSR2);
      // on récupère les valeurs de exit() des fils pour éviter les zombies...
```

```
      for (i=0 ; i<4 ; i++)
            wait(NULL);
      exit(0);
}

void recoit_fin_partie()
{
      printf("Joueur %d, score : %d\n", nb_joueur, score);
      exit(0);
}

int main()
{
      int   i, j;
      // Création de 4 pipes anonymes
      for (i=0 ; i<4 ; i++)
            pipe(p[i]);
      // initialisation du score pour tous
      score=0;
      // Création des 4 joueurs
      nb_joueur=0;
      while ( (nb_joueur<4) && ((tab_pid[nb_joueur]=fork())>0) )
            nb_joueur++;
      if (nb_joueur==4) // on a reçu 4x un nbre >0 au fork() => on est le père
      {
            signal(SIGUSR1, reception_patate_pere);
            // on ferme les pipes en réception :
            for (i=0 ; i<4 ; i++)
                  close(p[i][0]);
            // pour chacun des 4 fils, on envoie les 4 PID
            for (i=0 ; i<4 ; i++)
                  for (j=0 ; j<4 ; j++)
                        write(p[i][1], &(tab_pid[j]), sizeof(tab_pid[j]));
            // on ferme les pipes d'émission
            for (i=0 ; i<4 ; i++)
                  close(p[i][1]);
            // on envoie la patate ;-)
            i=num_joueur_aleat();
            kill(tab_pid[i], SIGUSR1);
            for (;;) ;
      }
      else
      {
            // code des fils (NB: chaque fils est le joueur nb_joueur)
            signal(SIGUSR1, reception_patate_fils);
            signal(SIGUSR2, recoit_fin_partie);
            // on ferme les pipes d'émission
            for (i=0 ; i<4 ; i++)
                  close(p[i][1]);
            // on reçoit les pid des joueurs
            for (i=0 ; i<4 ; i++)
                  read(p[nb_joueur][0], &(tab_pid[i]), sizeof(tab_pid[i]));
            // on ferme les pipes de réception :
            for (i=0 ; i<4 ; i++)
                  close(p[i][0]);
            for (;;) ;
      }
}
```

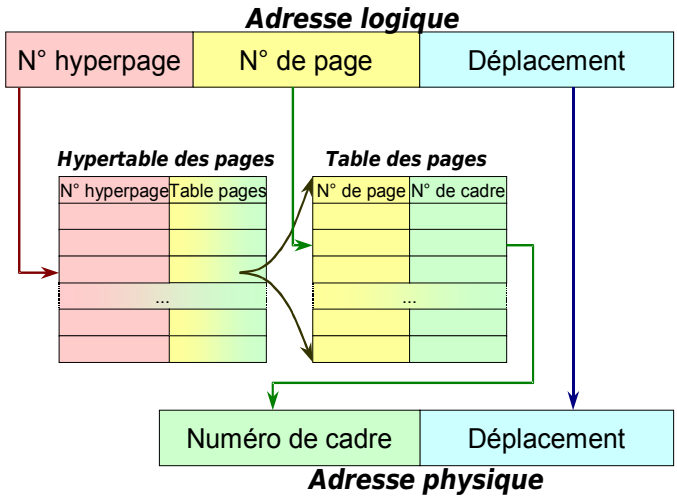
<u>Exercice n°2</u> : *mémoire, segmentation, pagination (3 questions pour un total de 6 points)*

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 : les threads *(3 questions pour un total de 6 points)*

Question 3.a) [*1,5 point*] : Le concept de thread permet d'effectuer de la multiprogrammation (plusieurs programmes se partagent le temps CPU), au même titre que les processus. Seulement, les processus souffrent de trois défauts majeurs :

- la création d'un nouveau processus (avec la création d'un nouveau PCB, la mise en œuvre de segments de mémoire isolés, etc.) est coûteuse, même si le « *copy on write* » permet d'alléger ce mécanisme ;
- le fait que deux processus aient deux espaces mémoire isolés rend la commutation de contexte coûteuse ;
- et pour la même raison (espaces mémoire isolés), les mécanismes de communication inter-processus sont eux aussi coûteux (recopie d'information par exemple).

Or, parfois, les programmeurs ont besoin de faire de la multiprogrammation, sans qu'il soit nécessaire (bien au contraire) que les deux tâches aient des espaces mémoire isolés. Les threads (ou processus légers) ont été inventés dans ce but : proposer un mécanisme qui permet de créer plusieurs tâches au sein d'un même processus. Ces tâches partagent le même code et les mêmes données (tas, variables globales, constantes, etc.). Seule la pile à l'exécution et l'état des registres sont spécifiques à chaque thread.

Question 3.b) [*1,5 point*] : Il existe deux types de thread : les threads utilisateur et les threads noyau.

Les threads utilisateurs sont implémentés dans une bibliothèque de fonctions qui s'exécutent en mode utilisateur. Le système d'exploitation (et en particulier l'ordonnanceur) n'a pas connaissance de la notion de thread. L'ordonnancement des différentes tâches des différents threads est assurée par un ordonanceur spécifique, qui s'exécute dans le processus en mode utilisateur (sans privilège système).

Les threads noyaux sont implémentés dans le système d'exploitation. L'ordonnanceur du système d'exploitation doit connaître la notion de thread, et doit gérer lui-même l'ordonnancement des tâches des différents threads au sein d'un processus.

Avantages des threads utilisateurs :

- ils permettent, dans les anciens systèmes d'exploitation, de proposer aux programmeurs le mécanisme de thread même quand le système d'exploitation ne proposait pas ce concept ;
- souvent, la bibliothèque de fonctions qui assure l'ordonancement des différents threads au sein d'un processus ne nécessite pas un coûteux passage en mode "privilégié".

Leur inconvénient : si un thread effectue un appel système bloquant, ce sont tous les threads du processus qui se retrouvent bloqués.

Question 3.c) [3 *points*] : Tout d'abord, le programmeur utilise les mutex **`Mut1`** et **`Mut2`** pour protéger l'accès à des variables locales. Or, ces variables sont stockées dans la pile à l'exécution. Sachant que chaque thread possède sa propre pile à l'exécution, ces variables ne sont pas partagées. D'ailleurs, le compilateur fournira une erreur " variables **`compteur_commun1`** et **`compteur_commun2`** inconues dans la fonction **`second_thread()`** ". La solution : ressortir les déclarations de ces deux variables dans le corps du programme, en dehors de toute fonction, afin que ces variables soient globales. Elles seront alors connues de toutes les fonctions, et leurs valeurs sera partagée entre les deux threads (d'où l'intérêt de les protéger par des mutex).

Ensuite, un des threads positionne un verrou `Mut1` avant de verrouiller `Mut2`. Inversement, le second thread verrouille `Mut2` avant `Mut1`. Il peut donc y avoir interblocage, ou **étreinte mortelle** entre les deux threads (s'il y a préemption d'un tread entre deux verrouillages, et que l'autre thread positionne ses deux verrous). Dans le cas présent, comme il n'y a que deux threads et deux ressources à protéger, la solution est simple : il suffit de toujours verrouiller les mutex dans le même ordre (par exemple en intervertissant les lignes " `pthread_mutex_lock(&Mut2);` " et " `pthread_mutex_lock(&Mut1);` " dans la fonction `second_thread()`.

Le code correct est le suivant (modifications en rouge) :

```
#include    <stdio.h>
#include    <stdlib.h>
#include    <unistd.h>
#include    <pthread.h>

extern      void gros_calcul();
extern      void petit_calcul();
extern      void modifie_compteurs(int *, int *);
extern      int on_continue(int);

pthread_mutex_t   Mut1=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t   Mut2=PTHREAD_MUTEX_INITIALIZER;

/* Variables globales (donc, visibles par tous les threads) */
int         compteur_commun1 = 0;
int         compteur_commun2 = 0;

void  *second_thread()
{
      while (on_continue(compteur_commun1))
      {
            gros_calcul();
            /* les mutex sont employés dans le même ordre : */
            pthread_mutex_lock(&Mut1);
            pthread_mutex_lock(&Mut2);
            modifie_compteurs(&compteur_commun1, &compteur_commun2);
            pthread_mutex_unlock(&Mut2);
            pthread_mutex_unlock(&Mut1);
      }
      pthread_exit(NULL);
      return(NULL);
}

int   main()
{
      pthread_t   th;
      /* les lignes ci-dessous ont été mises en commentaire */
      /* int            compteur_commun1 = 0; */
      /* int            compteur_commun2 = 0; */
      /* On crée le second thread */
      if ((pthread_create(&th, NULL, second_thread, NULL))!=0)
      {
            fprintf(stderr, "Erreur pthread_create()\n");
            exit(-1);
      }
```

```
	while (on_continue(compteur_commun1))
	{
		pthread_mutex_lock(&Mut1);
		pthread_mutex_lock(&Mut2);
		modifie_compteurs(&compteur_commun1, &compteur_commun2);
		pthread_mutex_unlock(&Mut2);
		pthread_mutex_unlock(&Mut1);
		petit_calcul();
		pthread_mutex_lock(&Mut1);
		compteur_commun1++;
		pthread_mutex_unlock(&Mut1);
	}
	pthread_join(th, NULL);
	pthread_mutex_destroy(&Mut1);
	pthread_mutex_destroy(&Mut2);
	return(0);
}

/* Fin du programme */
```