CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Samedi 13/09/2008
2 heures

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103

CORRECTION DE LA SESSION DE SEPTEMBRE
Année 2007 – 2008, deuxième semestre

Documents autorisés : NON

Cnam
CONSERVATOIRE NATIONAL
DES ARTS ET METIERS

Exercice n°1 : *ordonnancement temps-réel (3 questions pour un total de 8 points)*

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

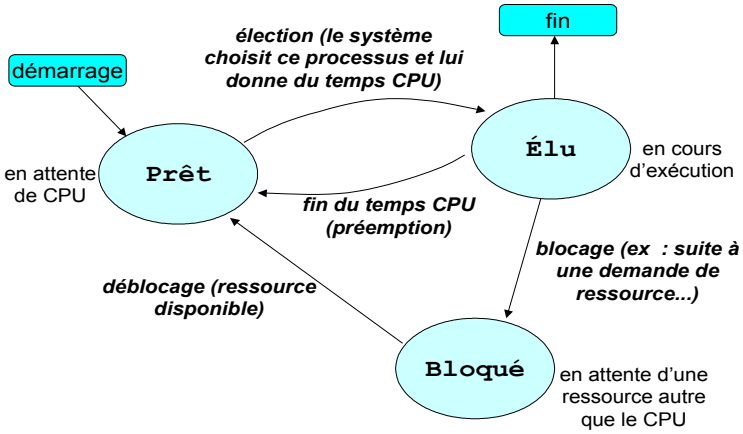


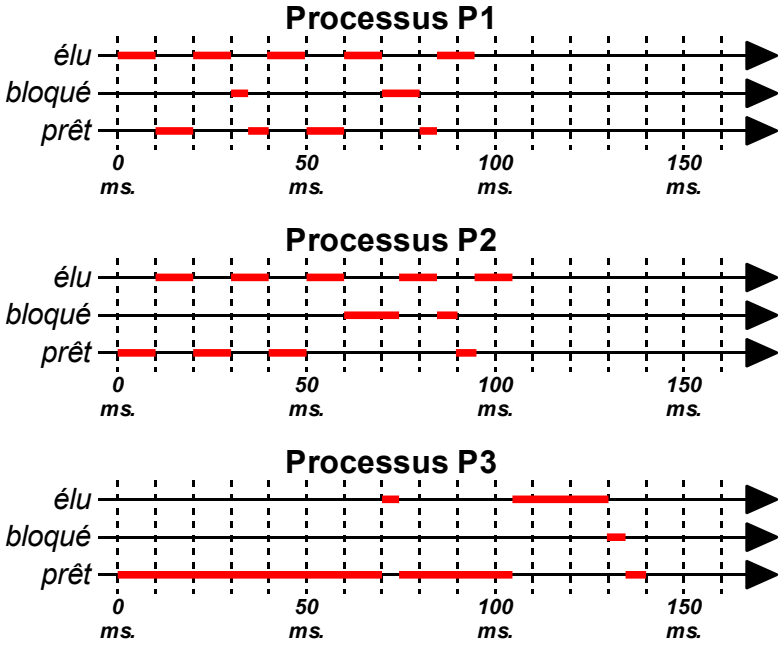
Diagramme d'état d'un processus Unix/Linux

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

L'algorithme du tourniquet (round robin) sous Linux utilisé pour la classe d'ordonnancement « `SCHED_RR` » élit le processus (ou un des processus) ayant la priorité **la plus élevée**. Lorsqu'un processus est élu, il s'exécute **durant un quantum de temps**, avant d'être préempté (sauf s'il a lancé une fonction système bloquant entre temps, ou si un processus de priorité plus élevé est élu).

Une file d'attente des processus de même priorité permet de lancer les processus les uns à la suite des autres pour un temps donné.

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



Temps d'exécution de P1 = 95 ms.

Temps d'exécution de P2 = 105 ms.

Temps d'exécution de P3 = 140 ms.

Exercice n°2 : threads *(4 questions pour un total de 8 points)*

Question 2.a) [*1,5 point*] : Les processus souffrent de trois défauts majeurs :

- la création d'un nouveau processus (avec la création d'un nouveau PCB, la mise en oeuvre de segments de mémoire isolés, etc.) est coûteuse, même si le « *copy on write* » permet d'alléger ce mécanisme ;
- la 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.). Seuls la pile à l'exécution et l'état des registres sont spécifiques à chaque thread.

Question 2.b) [*1 point*] : dans un thread, la protection d'une ressource pour s'en assurer l'exclusivité est réalisée avec les MUTEX (qui fonctionnent comme un verrou).

Question 2.c) [*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 fonction qui s'exécute 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.

Les threads utilisateurs ont été utilisés dans les anciens systèmes d'exploitation, quand ceux-ci ne connaissaient pas la notion de thread. Leur inconvénient : si un thread effectue un appel système bloquant, ce sont tous les threads du processus qui se retrouvent bloqués.

Question 2.d) [*4 points*] :

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


/* Ces deux variables doivent être globales pour que les 2 threads  */
/* puissent y accéder.                                              */
pthread_cond_t  Cond=PTHREAD_COND_INITIALIZER;
pthread_mutex_t Mut=PTHREAD_MUTEX_INITIALIZER;

extern void f1(); /* f1() et f2() sont définies ailleurs */
extern void f2();


/* Code du thread n°2 : */
void  code_thread_2()
{
      /* le thread n°2 se met en position d'attente */
      pthread_mutex_lock(&Mut);
      pthread_cond_wait(&Cond, &Mut);
```

```
      /* arrivé ici, la fonction f1() est terminée dans le thread n°1 */
      pthread_mutex_unlock(&Mut);
      /* on peut donc appeler f2() */
      f2();

      /* reste à faire le ménage : */
      pthread_mutex_destroy(&Mut);
      pthread_cond_destroy(&Cond);

      /* On quitte proprement le thread */
      pthread_exit(NULL);
      return(NULL);
}

/* Le corps de la fonction principale : */
int  main()
{
      pthread_t  th;
      if ((pthread_create(&th, NULL, code_thread_2, NULL))!=0)
      {
            fprintf(stderr, "Erreur pthread_create()\n");
            exit(-1);
      }

      /* On est le thread n°1, on exécute la fonction f1() */
      f1();

      /* f1() est terminée, on prévient le thread n°2 */
      pthread_mutex_lock(&Mut);
      pthread_cond_signal(&Cond);
      pthread_mutex_unlock(&Mut);

      /* On attend que le thread n°2 ait fini avant de tout quitter */
      pthread_join(th, NULL);
      return(0);
}
```

Exercice n°3 : la mémoire *(2 questions pour un total de 4 points)*

Question 3.a) [*2 points*] : La segmentation est une technique de *virtualisation* de la mémoire physique. Celle-ci est divisée en zones pouvant être **de différentes tailles**. Un segment est caractérisé par :

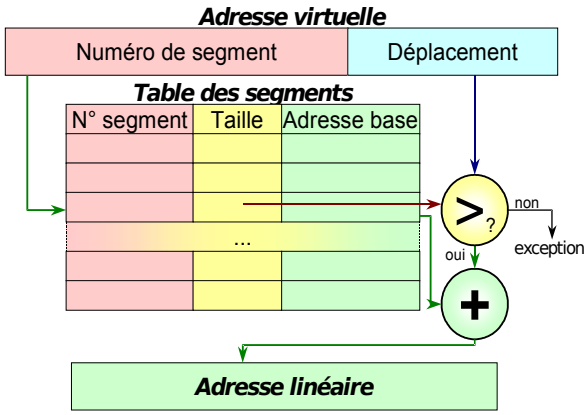
- l'adresse physique de sa base (adresse physique du début du segment),
- sa taille.

Une adresse virtuelle possède alors deux composantes : un numéro de segment, et un déplacement dans ce segment.

Le mécanisme physique du processeur qui gère ce mécanisme est ma « *MMU* » (*Memory Management Unit*, ou *Unité de Gestion de la Mémoire*). Elle contient une table des segments, qui est indexée par le numéro de segment, et qui, pour chaque entrée, contient :

- la taille du segment,
- et son adresse de base.

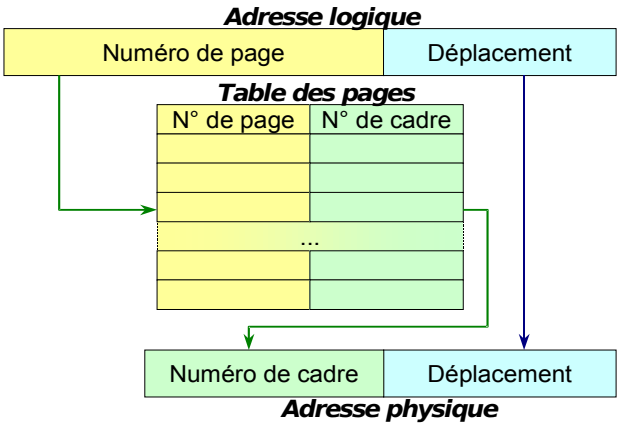
Pour traduire une adresse virtuelle, le MMU cherche l'entrée correspondant au numéro de segment dans la table des segments. Il vérifie que le déplacement contenu dans l'adresse virtuelle est inférieur à la taille du segment. Si c'est le cas, il additionne le déplacement à l'adresse de base du segment pour obtenir l'adresse physique :



Traduction d'adresse (segmentation)

Question 3.b) [*2 points*] : La pagination est aussi un mécanisme de *virtualisation* de la mémoire, qui travaille cette fois-ci sur des « morceaux » de mémoire de taille fixe, selon le principe suivant :

- les adresses mémoires émises par le processeur sont des adresses virtuelles linéaires, indiquant la position d'un mot dans la mémoire virtuelle ;
- cette mémoire virtuelle est formée de zones de même taille, appelées pages. Une adresse virtuelle est donc un couple (numéro de page, déplacement dans la page). La taille des pages est une puissance de deux (en général, entre 512 et 8'192 octets), de façon à déterminer sans calcul le déplacement dans la page (exemple : 10 bits de poids faible de l'adresse virtuelle pour des pages de 1024 mots), et le numéro de page (les autres bits) ;
- la mémoire physique est également composée de zones de même taille, appelées « cadres » (frames en anglais), dans lesquelles prennent place les pages. A noter que la taille d'un cadre est égale à la taille d'une page ;
- le MMU assure la conversion des adresses virtuelles en adresses physiques, en consultant une « **table des pages** » (page table en anglais) pour connaître le numéro du cadre qui contient la page recherchée. L'adresse physique obtenue est le couple (numéro de cadre, déplacement) ;
- il peut y avoir plus de pages que de cadres : les pages qui ne sont pas en mémoire sont stockées sur un autre support (disque dur), elles seront ramenées dans un cadre si besoin.



Traduction d'adresse (pagination)