Durée : 2 heures

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103 - NANCY/METZ

CORRECTION DE L'EXAMEN
Année 2009 – 2010, deuxième semestre
Coefficient : 2/3

Documents autorisés : NON

Cnam
CONSERVATOIRE NATIONAL
DES ARTS ET METIERS

<u>Exercice n°1</u> : synchronisation *(3 questions pour un total de 6 points)*

Question 1.a) [*2 points*] : Un sémaphore est un objet (ou type abstrait de donnée) contenant une variable entière positive et des mécanismes qui permettent de gérer de façon esthétique les accès concurrent à des ressources par plusieurs processus, en bloquant les processus (sans consommation de CPU) lorsque la ressource n'est pas accessible.

La fonction `Init(S, val)` permet d'initialiser le sémaphore `S` en lui attribuant la valeur `val`.

La fonction `P(S)` : si `S` est strictement supérieur à zéro, `S` est décrémenté. Sinon, le processus appelant est bloqué (pas de consommation de CPU), en attendant qu'un autre processus fasse une opération `V(S)`.

La fonction `V(S)` débloque un des éventuels processus bloqué par un `P(S)` avec `S` nul, et incrémente `S` dans le cas contraire.

À noter que les fonctions `P()` et `V()` sont des fonctions unitaires (il ne peut y avoir préemption durant leur exécution).

Question 1.b) [*2 points*] : Le sémaphore `m` est un verrou (ou mutex) qui permet de s'assurer l'accès exclusif à la variable `nb_Voit_Att`.

Le sémaphore `lavage` est à 1 si la station de lavage est disponible, et à 0 si elle est en cours de lavage d'un véhicule. Permet de bloquer les voitures qui souhaitent entrer dans la station de lavage si elle est déjà en service.

Le sémaphore `voiture` contient le nombre de voitures en attente d'être lavées (bloquées sur le parking ou en instance de rentrer dans la station de lavage). Permet de bloquer la station de lavage lorsqu'il n'y a pas de voiture en attente d'être lavée.

Le sémaphore `fin_lavage` passe à 0 lorsque la voirure entre dans son cycle de lavage (bloquant alors cette dernière), et passe à 1 après le coup de polish (débloquant alors la voiture pour qu'elle sorte).

Question 1.c) [*2 points*] : Deux mécanismes empêchent que la variable `nb_Voit_Att` soit négative :

- la variable est décrémentée par la fonction `station_de_lavage()` après un `P(voiture)`. Or, pour que ce `P(voiture)` ne soit pas bloquant, il faut qu'au moins une voiture ait fait un `V(voiture)` au préalable. Or, ce `V(voiture)` n'à lieu qu'après l'incrémentation de la variable `nb_Voit_Att`. Autrement dit, la décrémentation de cette variable ne peut avoir lieu qu'après au minimum une incrémentation ;
- de plus, le verrou `m` assure l'exclusivité de l'accès à cette variable, évitant ainsi tout effet de bord lié à un changement simultané de sa valeur par plusieurs processus.

* * * * * *

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

Question 2.a) [*2 points*] : La **pagination** permet la mise en œuvre de la mémoire virtuelle.

Elle est gérée par le **MMU** (Memory Management Unit), ou Unité de Gestion de la Mémoire.

Lorsqu'il n'y a plus de place en mémoire centrale, une case est choisie pour être *swapée* (copiée sur disque pour libérer de la place en RAM). Or, cette case a pu déjà être *swapée* par le passé.

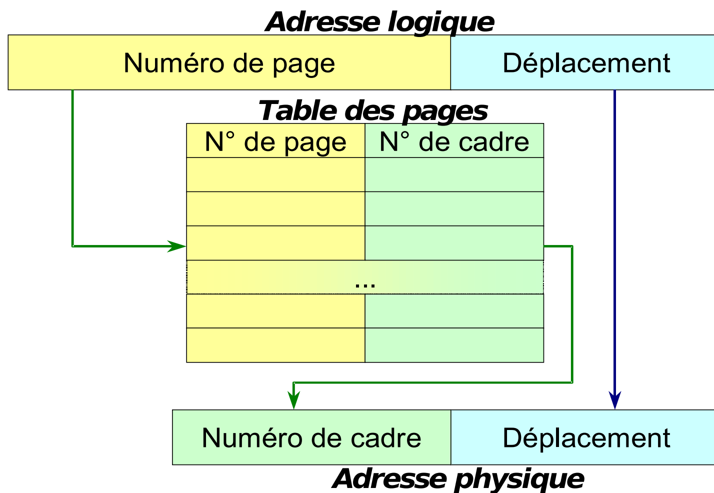
Dans ce cas, il est tout à fait intéressant de savoir si le contenu de la case a été modifié en mémoire centrale depuis son dernier chargement.

Ainsi, si le contenu n'a pas été modifié (cas où les accès par les processus ont été réalisés en lecture seule), et si la zone de swap qui contenait cette case n'a pas été ré-attribuée pour stocker une autre case, il est inutile de recopier son contenu sur disque.

C'est pour cette raison que la table des pages contient aussi un « bit M », dit bit de *Modification* (ou « bit D » pour *Dirty* en Anglais). Ce bit vaut 1 si la page est nouvellement allouée (vu qu'elle n'a jamais été swapée par le passé), ou lorsque le contenu de la case a été modifié depuis son dernier chargement en provenance de la zone de swap. Et il vaut 0 après un chargement du cadre depuis un bloc de swap, et reste à 0 tant que le contenu du cadre n'est pas modifié en RAM.

Avec ce mécanisme, si ce cadre est élu pour être à nouveau stocké sur le disque, si le bloc de swap n'a pas été réutilisé et que le bit M vaut 0, il n'y aura pas besoin de recopier le cadre sur le disque (les contenus étant identiques).

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



Traduction d'adresse (pagination)

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	***3***	***4***	***5***	***6***	***4***	***7***	***4***	***0***	***6***	***7***	***3***	***7***	***6***	***5***	***6***	***4***	***5***	***3***	***4***	***6***	***5***	***4***
Case 0	3	3	**3**	6	6	6	**6**	0	0	**0**	3	3	**3**	5	5	5	5	5	**5**	6	6	6
Case 1		4	4	4	4	4	4	4	**4**	7	7	7	7	7	**7**	4	4	4	4	4	4	4
Case 2			5	5	**5**	7	7	**7**	6	6	6	6	6	6	6	6	**6**	3	3	**3**	5	5
Défaut de page	O	O	O	O	N	O	N	O	O	O	O	N	N	O	N	O	N	O	N	O	O	N

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

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

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

* * * * * *

<u>Exercice n°3</u> : les threads *(3 questions pour un total de 8 points)*

Question 3.a) [*3,5 points*] :

Dans un système d'exploitation multiprogrammé, un **processus** est un programme qui s'exécute sans aucun privilège lui permettant d'accéder directement aux périphériques (il a obligation de passer par des primitives systèmes pour réaliser ses opérations d'entrées/sorties), et dont l'espace mémoire est totalement isolé de celui des autres processus. Les processus partagent le CPU (c'est l'ordonnanceur qui assure cette tâche au niveau du système d'exploitation).

Durant sa vie, un processus se retrouve régulièrement dans les trois états suivants :

- **prêt** : le processus est en attente de CPU ;
- **élu** : il est en cours d'exécution ;
- **bloqué** : le processus est en attente d'un événement (entrée/sortie, signal, etc.).

Le passage dans ces trois états peut se schématiser ainsi :

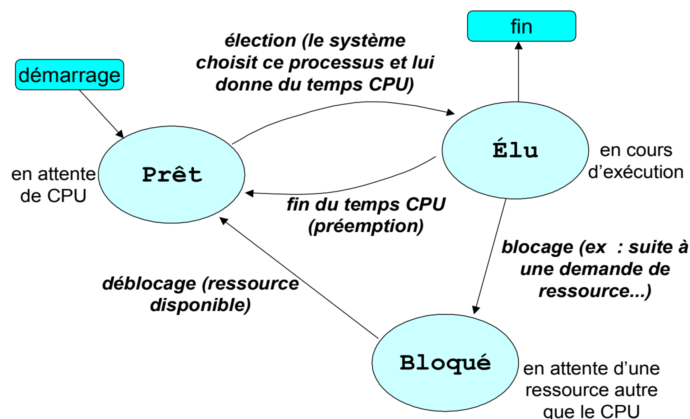


Diagramme d'état d'un processus Unix/Linux

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. L'ordonnancement des threads est moins coûteux en terme de charge CPU et de recopie mémoire que les changements de contexte des processus.

Question 3.b) [*1 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 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) [*3,5 points*] :

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

extern void      petit_calcul(), gros_calcul();

pthread_cond_t    Cond=PTHREAD_COND_INITIALIZER;
pthread_mutex_t   Mut=PTHREAD_MUTEX_INITIALIZER;

void  *second_thread()
{
      petit_calcul();
      pthread_mutex_lock(&Mut);
      pthread_cond_wait(&Cond, &Mut);
      pthread_mutex_unlock(&Mut);
      pthread_mutex_destroy(&Mut);
      pthread_cond_destroy(&Cond);
      pthread_exit(NULL);
}

int   main()
{
      pthread_t   th;
      if ((pthread_create(&th, NULL, second_thread, NULL))!=0)
      {
            fprintf(stderr, "Erreur pthread_create()\n");
            exit(-1);
      }
      gros_calcul();
      pthread_mutex_lock(&Mut);
      pthread_cond_signal(&Cond);
      pthread_mutex_unlock(&Mut);
      pthread_join(th, NULL);
      return(0);
}
/* Fin du programme */
```