CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Durée : 2 heures

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103 - NANCY/METZ

CORRECTION DE L'EXAMEN DU 28 JUIN 2011

Année 2010 – 2011, deuxième semestre
Coefficient : 2/3

Documents autorisés : NON

Cnam

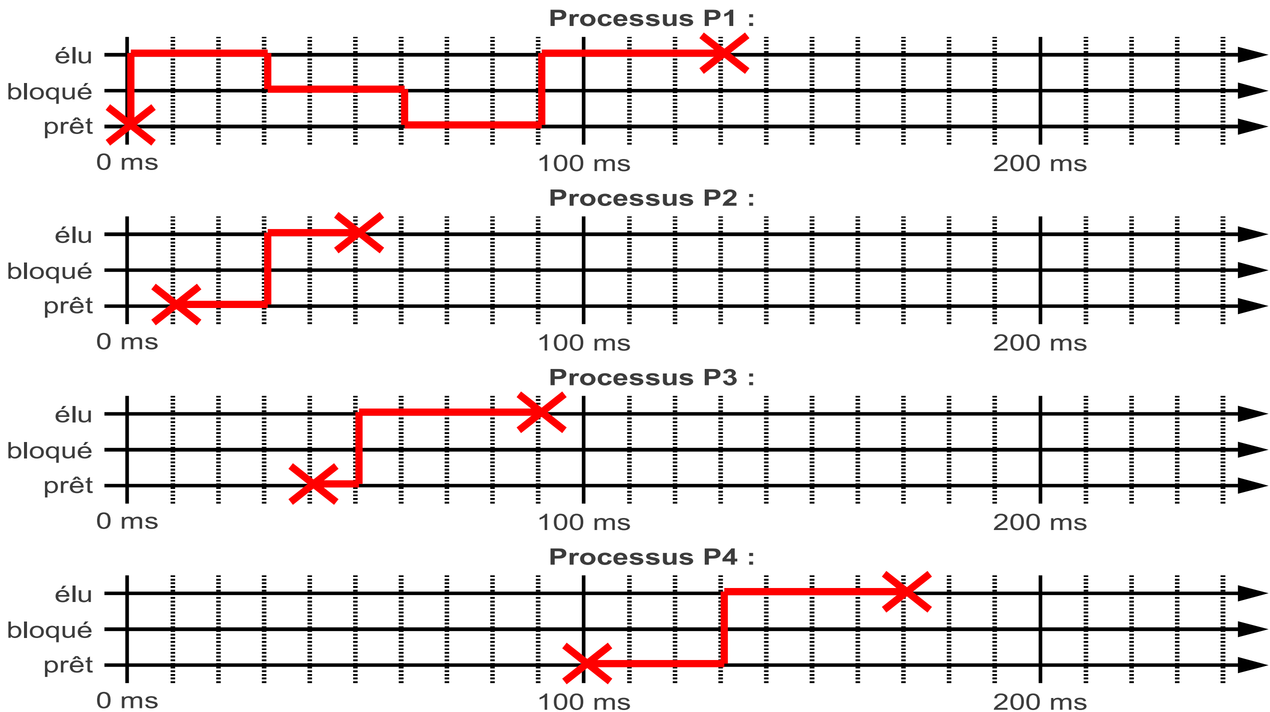
CONSERVATOIRE NATIONAL
DES ARTS ET METIERS

Exercice n°1 : synchronisation *(3 questions pour un total de 5 points)*

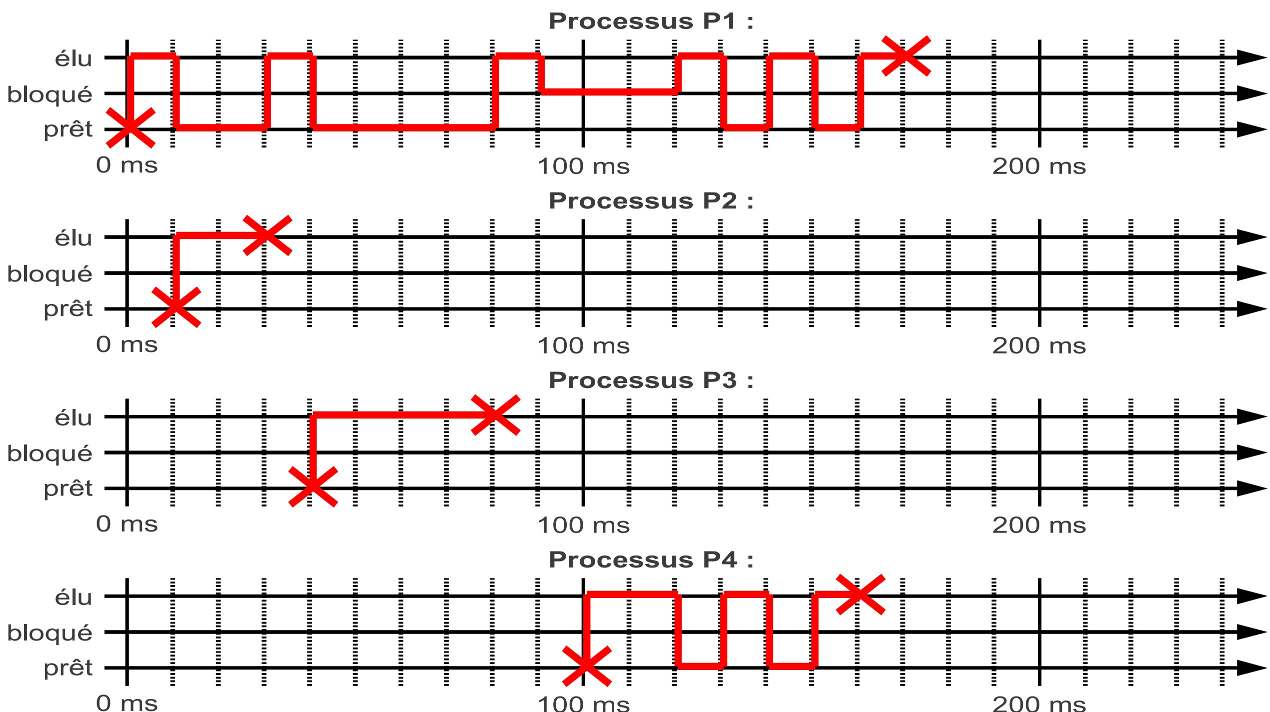
Question 1.a) [*1 point*] : L'ordonnanceur (ou *scheduler* en anglais) est le code du système d'exploitation qui réalise le travail d'élection (c'est à dire qui choisit le processus – ou le thread si cette notion est connue du système d'exploitation – qui sera exécuté, lorsque plusieurs processus – ou thread – sont en compétition pour obtenir le CPU – état prêt –). Dans un système d'exploitation préemptif, il arme une horloge pour partager équitablement le CPU (afin d'éviter les famines). Enfin, l'ordonnanceur effectue le travail de commutation de contexte.

Note du correcteur : ½ point si le candidat explique le travail d'élection, ½ point supplémentaire s'il aborde la notion de thread, et 1 point pour le travail de commutation de contexte.

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



Question 1.c) [2 points] : (note au correcteur : 2 solutions possibles à la fin, P1/P4... ou P4/P1...)



Exercice n°2 : *contrôle d'un capteur de pression d'une chaîne de montage (4 questions pour un total de 8 pts)*

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

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

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

Premier scénario : le processus fils envoie le signal `SIG_USR2` à son père suite à l'appui sur le bouton par le technicien. Puis, hasard du programme, il y a préemption. Le CPU est donné au programme principal, qui a reçu le signal `SIG_USR2`. Dans ce cas, la fonction `maintenance_ok()` est exécutée. Puis, le CPU est redonné au programme fils, le technicien appuie une seconde fois sur le bouton. Le signal `SIG_USR2` est envoyé une seconde fois, et la fonction `maintenance_ok()` est exécutée une seconde fois. La variable « `nb_utilisation_capteur` » est ainsi incrémentée deux fois !

Second scénario : le processus fils envoie le signal `SIG_USR2` deux fois au père à l'affilée (sans que le *scheduler* ne lui redonne le CPU entre ces deux envois de signal). Comme nous utilisons des **signaux classiques**, il n'y a pas empilement des signaux. Lorsque le père sera de nouveau élu, la fonction `maintenance_ok()` ne sera exécutée qu'une seule fois.

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

S'il peut y avoir plusieurs processus qui viennent lire `nb_utilisation_capteur`, 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 2.d) [*4 points*] :

```
/* variables globale : */
int nb_utilisation_capteur = 3;
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_utilisation_capteur, et le sémaphore 1 permet d'assurer l'exlusivité */
/* sur la variable nb_utilisation_capteur */
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_utilisation_capteur(int n)
{
    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_utilisation_capteur = nb_utilisation_capteur + n;
    ret = nb_utilisation_capteur;
    /* 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_utilisation_capteur()
{
    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_utilisation_capteur; /* 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 au 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°3 : les threads *(3 questions pour un total de 7 points)*

Question 3.a) [*3 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 :

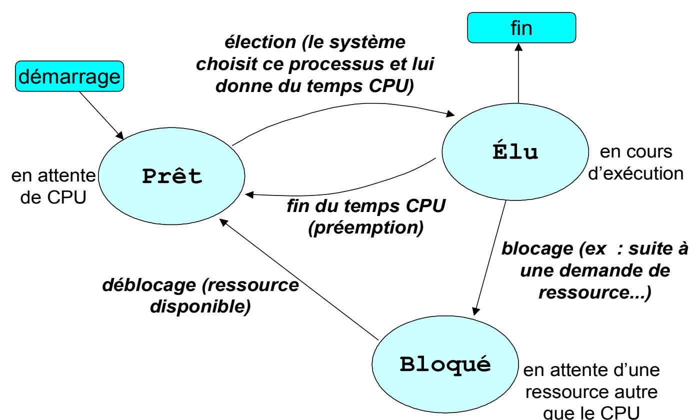


Diagramme d'état d'un processus Unix/Linux

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 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 */
```