

Durée : 2 heures

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103

EXAMEN

Année 2008 – 2009, deuxième semestre

Coefficient : 2/3

Documents autorisés : NON

Exercice n°1 : thread (3 questions pour un total de 6 points)

Un programmeur propose le code source suivant :

```
#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;

void *second_thread()
{
    while (on_continue(compteur_commun1))
    {
        gros_calcul();
        pthread_mutex_lock(&Mut2);
        pthread_mutex_lock(&Mut1);
        modifie_compteurs(&compteur_commun1, &compteur_commun2);
        pthread_mutex_unlock(&Mut1);
        pthread_mutex_unlock(&Mut2);
    }
    pthread_exit(NULL);
    return(NULL);
}

int main()
{
    pthread_t th;
    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 */
```

Dans ce programme, les fonctions `gros_calcul()` et `petit_calcul()` effectuent quelques calculs pendant une durée indéterminée.

La fonction `on_continue()` retourne 0 quand le programme doit se terminer. Rien ne nous permet de savoir à l'avance combien de fois cette fonction sera appelée avant de retourner un nombre nul qui permettra aux deux threads de sortir de leur boucle.

Enfin, la fonction `modifie_compteurs()` modifie les deux variables `compteur_commun1` et `compteur_commun2`.

Question 1.a) [1,5 point] : Qu'est-ce qu'un thread ? Quel est l'avantage d'un thread par rapport à un processus ?

Question 1.b) [1,5 point] : Quels sont les deux types de thread ? Pour chacun d'eux, précisez les avantages/inconvénients.

Question 1.c) [3 points] : Dans ce code, le programmeur a effectué deux erreurs conceptuelles fondamentales.

Une d'entre elles sera d'ailleurs détectée par le compilateur. La seconde pourra engendrer un comportement non désiré.

Quelles sont ces deux erreurs, et proposez une correction au code pour les éviter.

Exercice n°2 : ordonnancement temps-réel (3 questions pour un total de 7 points)

Sur une station de travail monoprocesseur tournant sous Linux, trois processus temps-réel P1, P2, et P3 se partagent les ressources de temps de calcul du CPU. Tous 3 appartiennent à la classe d'ordonnancement « **SCHED_RR** » (l'ordonnancement s'effectue avec l'algorithme du tourniquet – *round robin* – associé à une priorité). Les processus P1 et P2 ont une priorité de 50, et le processus P3 a une priorité de 40 (une plus grande valeur correspond à une priorité **la plus forte**). Sur cette version du système d'exploitation, le quantum de temps pour l'algorithme du tourniquet a été fixé à 10 ms. On supposera le temps de commutation de contexte comme négligeable.

En plus de leur besoin en temps de calcul, chacun des trois processus réalisent aussi des entrées/sorties à destination du disque dur. Ces opérations d'entrées/sorties bloquent le processus et ne sont pas consommatrices de CPU.

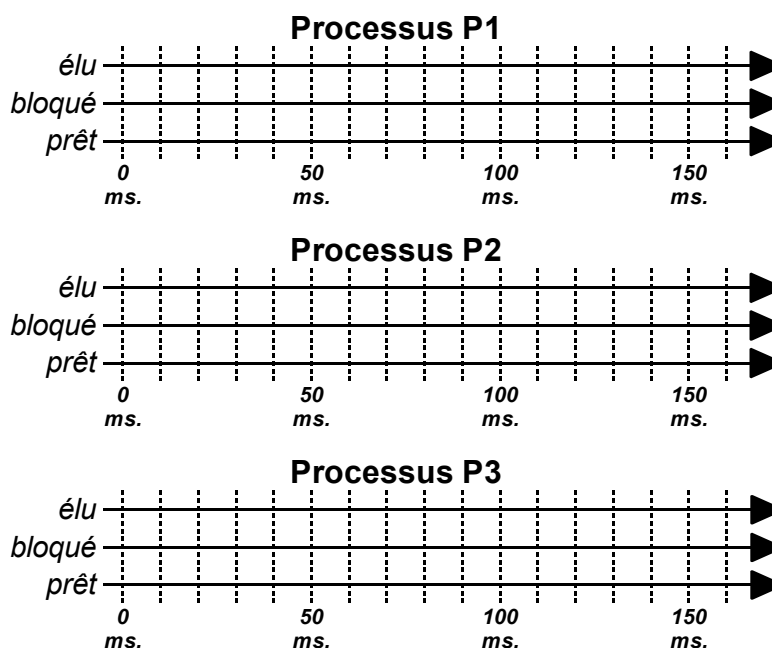
Les opérations que doivent réaliser ces trois processus sont indiquées dans le tableau suivant :

Processus P1 (priorité 50)	Processus P2 (priorité 50)	Processus P3 (priorité 40)
20 ms. de temps de calcul	30 ms. de temps de calcul	30 ms. de temps de calcul
5 ms. d'entrées/sorties	15 ms. d'entrées/sorties	5 ms. d'entrées/sorties
20 ms. de temps de calcul	10 ms. de temps de calcul	5 ms. de temps de calcul
10 ms. d'entrées/sorties	5 ms. d'entrées/sorties	
10 ms. de temps de calcul	10 ms. de temps de calcul	

Question 1.a) [1,5 point] : dessinez le schéma des trois principaux états que peuvent prendre les processus Linux pendant leur exécution, et indiquer les raisons qui peuvent faire passer un processus d'un état à un autre.

Question 1.b) [1,5 point] : rappelez brièvement le fonctionnement de l'algorithme du tourniquet associé à une priorité, utilisé dans la classe d'ordonnancement « **SCHED_RR** » sous Linux.

Question 1.c) [4 points] : sachant que les trois processus sont lancés en même temps, remplissez le chronogramme d'exécution (selon le modèle ci-dessous) des trois processus indiquant pour chacun s'il est dans l'état « prêt », « bloqué », ou « élu ». Donnez pour chaque processus son temps de réponse (temps écoulé entre le début et la fin de l'exécution du processus, vu de l'utilisateur). Rq : *ne pas écrire sur le sujet, faire les schémas sur votre copie.*



Exercice n°3 : sculpteur (une question pour un total de 7 points)

La recherche spatiale a besoin de réaliser de tous petits objets en céramique sur mesure. Pour ce, les chercheurs ont élaboré un automate capable de sculpter ces objets selon la forme voulue, à partir de petits cubes de céramique. L'automate « use » petit à petit le cube pour obtenir in fine la forme voulue. Cette usure est réalisée à l'aide de trois actions : une action chimique, une action mécanique (polissage), et une action de sublimation à l'aide d'un laser.

Même si toute opération de sculpture commence par un traitement chimique à l'acide, il est généralement impossible de savoir à l'avance dans quel ordre et combien de fois chacune de ces trois actions doit être réalisée.

L'automate commence par réduire le cube à l'aide d'actions chimiques. A la fin de cette opération, un capteur intégrant le module chimique décide s'il faut mener un limage au laser ou un polissage mécanique. Si c'est le laser qui est choisi, à la fin de l'opération, les capteurs du laser décident s'il faut arrêter, recommencer un traitement chimique, ou un polissage. De même, après un polissage mécanique, un capteur détecte si l'objet est terminé, ou s'il faut recommencer un traitement chimique ou au laser.

Pour résumer : chacune des trois actions (arrêt, ou autre traitement) peut être décidée par chaque module quand il a terminé sa tâche, et l'enchaînement des traitements est indécidable à l'avance.

Question 3.a) [7 points] : programmez en C le code qui réalise cet automate :

- le programme principal lance 3 processus fils : un fils qui sera appelé par la suite « chimique », un fils qui sera appelé « laser », et un fils qui sera appelé « mécanique » ;
- à l'aide d'un pipe anonyme, le programme diffuse à chacun des fils le PID de ses processus frères (par exemple, il envoie en fils « chimique » au minimum le PID du processus « laser », ainsi que le PID du processus « mécanique ») ;
- chacun des trois fils est endormi sans consommation de CPU via l'appel à la primitive `pause()` ;
- le père envoie un signal utilisateur `SIGUSR1` classique au fils « chimique », puis s'endort grâce à la fonction `pause()`. Ce signal réveille le fil « chimique ». Ce dernier lance alors une fonction `action_chimique()` qui effectue l'action demandée. Cette fonction retourne un entier qui indiquera quelle action doit être menée ensuite :
 - -1 : le processus envoie un signal utilisateur `SIGUSR1` au processus « laser » puis s'endort de nouveau ;
 - 0 : le processus envoie un signal utilisateur au processus parent, puis s'endort ;
 - 1 : le processus envoie un signal utilisateur au processus « mécanique » puis s'endort de nouveau ;
- à la réception d'un signal utilisateur `SIGUSR1`, le processus « laser » est réveillé. Il appelle une fonction `action_laser()` qui retournera elle aussi -1 s'il faut un nouveau traitement chimique (envoi d'un signal `SIGUSR1` au processus « laser »), 0 si la pièce est terminée (envoi d'un signal `SIGUSR1` au processus parent), et 1 s'il faut un nouveau traitement mécanique (envoi d'un signal `SIGUSR1` au processus « mécanique ») ;
- idem, à la réception d'un signal `SIGUSR1`, le processus « mécanique » se réveille et appelle une fonction `action_mecanique()` qui retournera -1 s'il faut un nouveau traitement chimique, 0 pour signaler la fin de la pièce au père, ou 1 s'il faut un traitement au laser ;
- lorsque le processus père reçoit un signal utilisateur, il envoie un signal `SIGKILL` aux trois fils, libère les ressources devenues inutiles dans le système, et quitte proprement par l'appel à la fonction `exit()`.

Vous n'avez pas à programmer les fonctions « `int action_chimique();` », « `int action_laser();` », et « `int action_mecanique();` ». Le traitement des codes d'erreur suite aux appels aux primitives système pourront être des plus simples.

Annexe : les appels systèmes utiles (extraits des « man » Linux)

- `int socket(int domain, int type, int protocol)` : crée un point de communication, et renvoie un descripteur (ou `-1` si erreur). Ici, domain doit être `PF_INET`, le type `SOCK_DGRAM` (protocole UDP), le protocole à 0.
- `int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen)` : fournit à la socket sockfd, l'adresse my_addr, adresse de longueur addrlen octets. Retourne 0 si OK, `-1` sinon. Avec :

```
struct sockaddr_in {
    short sin_family;          /* AF_INET */
    u_short sin_port;         /* Num. de port en endian Internet */
    struct in_addr sin_addr;  /* Adresse IP (en endian Internet) */
    char sin_zéro [8];        /* Bourrage */
};
struct in_addr {
    u_long s_addr;
};
```
- `int recvfrom(int s, void *buf, int len, int flags, struct sockaddr *from, int *fromlen)` : utilisé pour recevoir des messages depuis un socket. Avec s le descripteur de socket, buf une zone mémoire de taille minimum len, len le nombre d'octets demandés, from est rempli par l'adresse source, et fromlen la taille (en octets) de cette adresse source. Retourne le nombre d'octets reçus, `-1` en cas d'erreur.
- `int sendto(int s, const void *buf, int len, int flags, const struct sockaddr *to, int tolen)` : envoie dans le socket s les len octets pointés par buf, à destination de l'adresse to de longueur tolen. Ici, flags vaut 0. Retourne le nombre d'octets écrits, `-1` en cas d'erreur.
- `bzero(void *ptr, int n)` : mets à 0 les n octets pointés par ptr.
- `memcpy(void *dest, void *source, int n)` : copie les n octets pointés par source vers dest.
- `memcmp(void *ad1, void *ad2, int n)` : retourne 0 si les n octets pointés par ad1 sont égaux à ceux pointés par ad2, et un nombre non nul sinon.
- `int htons(int n)` : normalise l'entier n au format réseau.
- `int htonl(long l)` : normalise l'entier long l au format réseau.
- `int ntohs(int n)` : normalise l'entier n au format local.
- `int ntohl(long l)` : normalise l'entier long l au format local.
- `struct hostent *gethostbyname(const char *name)` : retourne l'adresse IP de la machine de nom name, dans une structure

```
struct hostent {
    char *h_name;          /* Nom officiel de l'hôte. */
    char **h_aliases;     /* Liste d'alias. */
    int h_addrtype;       /* Type d'adresse de l'hôte. */
    int h_length;         /* Longueur de l'adresse. */
    char **h_addr_list;   /* Liste d'adresses. */
};
#define h_addr h_addr_list[0] /* pour compatibilité. */
```
- `int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)` : permet à un programme de surveiller des descripteurs de fichiers contenus dans les ensembles readfds (ensemble de descripteurs à surveiller pour voir si des octets sont disponibles en lecture), writefds (ensemble de descripteurs à surveiller pour voir si des octets sont disponibles en écriture), exceptfds (recherche des conditions exceptionnelles). nfds est égal à 1 + numéro du descripteur le plus grand des 3 ensembles readfds, writefds, et exceptfds. timeout permet de gérer un timer, et est de type :

```
struct timeval {
    long tv_sec;          /* secondes */
    long tv_usec;        /* microsecondes */
};
```

La fonction retourne le nombre de descripteurs ayant une action possible, 0 en cas de timeout. Quatre macros sont disponibles pour la manipulation ensembles : « void FD_ZERO(fd_set *set); » efface un ensemble. « void FD_SET(int fd, fd_set *set); » et « void FD_CLR(int fd, fd_set *set); » ajoutent et suppriment un descripteur dans un ensemble. « int FD_ISSET(int fd, fd_set *set); » vérifie si un descripteur est contenu dans un ensemble.

- int pipe(int **filedes**[2]) : crée une paire de descripteurs de tube, et les place dans un tableau **filedes** (**filedes**[0] est utilisé pour la lecture, et **filedes**[1] pour l'écriture). Retourne -1 si erreur, 0 sinon.
- int read(int **fd**, void ***buf**, size_t **count**) : lit count octets depuis le fichier **fd** et place les octets lus dans **buf**. Retourne le nombre d'octets lus, -1 sinon.
- int write(int **fd**, const void ***buf**, size_t **count**) : écrit dans le fichier **fd** **count** octets pointés par **buf**. Retourne le nombre d'octets écrits, -1 sinon.
- close(int **fd**) : ferme proprement le descripteur de fichier **fd**.
- void (*signal(int **signum**, void (***handler**)(int)))(int) : positionne la fonction **handler** comme fonction appelée en cas de réception du signal **signum**.
- int fork() : crée un processus fils en copiant le processus appelant. Retourne une valeur négative en cas d'erreur, 0 au fils, et le PID du processus nouvellement créé au père.
- int wait(int ***status**) : attend la fin d'un enfant. Si **status** est non NULL, met dans ***status** le code retourné par cet enfant. Retourne un nombre négatif en cas d'erreur, ou le PID du fils qui vient de se terminer. La lecture du statut supprime l'entrée du processus fils de la table des processus du système.
- int pause() : blocage d'un processus (sans consommation de CPU) dans l'attente d'un signal. Cette fonction renvoie toujours une valeur -1.
- int kill(pid_t **pid**, int **sig**) : envoie le signal **sig** au processus numéro **pid**. Retourne un nombre négatif en cas d'erreur.
- int getpid() : permet à un processus de connaître son propre PID. Retourne un nombre négatif en cas d'erreur.
- int getppid() : permet à un processus de connaître le PID de son père. Retourne un nombre négatif en cas d'erreur.
- int msgget(key_t **key**, int **msgflg**) : renvoie l'identifiant de la file de messages associée à la clé **key**. Dans les problèmes énoncés ici, **msgflg** vaudra : **IPC_CREAT | S_IRUSR | S_IWUSR**
- msgctl(**msqid**, IPC_RMID, NULL) : permet de détruire la file de message d'identifiant **msqid**.
- int msgsnd(int **msqid**, const void ***msgp**, size_t **msgsz**, int **msgflg**) : envoie sur la file ayant comme identifiant **msqid** un message pointé par **msgp**. Ce message doit avoir la forme d'une structure du type :

```

struct msgbuf {
    long      mtype;      /* type de message ( > 0 ) */
    mon_message message; /* contenu du message */
};

```

msgsz est la taille du type "mon_message", et **msgflg** vaudra 0 dans notre cas. La fonction retourne 0 si tout va bien, -1 sinon.

- `int msgrcv(int msgid, const void *msgp, size_t msgsz, long msgtyp, int msgflg)` : reste bloqué sur la file ayant comme identifiant **msgid** jusqu'à l'arrivée d'un message pointé par **msgp**. Ce message doit avoir la forme d'une structure du type :

```
struct msgbuf {
    long      mtype;      /* type de message ( > 0 ) */
    mon_message message; /* contenu du message */
};
```

msgsz est la taille du type "mon_message", **msgflg** vaudra 0 dans notre cas. Si l'argument **msgtyp** vaut 0, le premier message est lu (quel que soit son type). Si **msgtyp** est supérieur à 0, alors seul le premier message de type **msgtyp** est extrait de la file. La fonction retourne -1 en cas d'erreur, ou le nombre d'octets écrits dans le champ "**msgp.message**".
- `int semget(key_t key, int nsems, int semflg)` : Cette fonction retourne l'identifiant de l'ensemble de sémaphores associé à la valeur de clé **key**. La valeur de **nsems** indique le nombre de sémaphores que doit contenir cet ensemble. Dans les problèmes énoncés ici, **semflg** vaudra : `IPC_CREAT|S_IRUSR|S_IWUSR`
- `semctl(int sem_id, int sem_num, SETVAL, int v0)` : initialise le sémaphore numéro **sem_num** de l'ensemble **sem_id** à la valeur **v0**.
- `int semop(int semid, struct sembuf *sops, unsigned nsops)` : fonction qui effectue des opérations sur les membres de l'ensemble de sémaphores identifié par **semid**. Chacun des **nsops** éléments dans le tableau pointé par **sops** indique une opération à effectuer sur un sémaphore en utilisant une structure `struct sembuf` contenant les membres suivants :

```
unsigned short sem_num; /* Numéro du sémaphore */
short      sem_op; /* Opération sur le sémaphore */
short      sem_flg; /* Options pour l'opération */
```

Les options possibles pour **sem_flg** sont `IPC_NOWAIT` (ne sera pas utile ici) et `SEM_UNDO`. Si une opération indique l'option `SEM_UNDO`, elle sera annulée lorsque le processus se terminera. L'ensemble des opérations contenues dans le tableau **sops** est effectué atomiquement. Les opérations sont toutes réalisées en même temps, et seulement si elle peuvent toutes être effectuées.
Chaque opération est effectuée sur le **sem_num**-ième sémaphore de l'ensemble. Le premier sémaphore est le numéro 0. Pour chaque sémaphore, l'opération est l'une des 3 décrites ci-dessous :
 - Si l'argument **sem_op** est un entier positif, la fonction ajoute cette valeur au **sem_num**-ième sémaphore. Cette opération n'est jamais bloquante.
 - Si **sem_op** vaut 0, le processus attend que **semval** soit nul ; si **semval** vaut zéro, l'appel système continue immédiatement.
 - Si **sem_op** est inférieur à 0, si **semval** est supérieur ou égal à la valeur absolue de **sem_op**, la valeur absolue de **sem_op** est soustraite du **sem_num**-ième sémaphore. Sinon, l'appel est bloquant.
En cas de réussite, `semop()` renvoie 0, et -1 sinon.
- `semctl(sem_id, 0, IPC_RMID, 0)` : permet de détruire l'ensemble de sémaphores identifié par **sem_id**.
- `int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)` : permet la création d'un nouveau thread, identifié par l'identificateur **thread**, et attaché à l'exécution de la routine (**start_routine**). Le fil d'exécution démarre son exécution au début de la fonction **start_routine** spécifiée et disparaît à la fin de l'exécution de celle-ci. L'argument **attr** sera mis à NULL dans les exercices ici. Enfin, **arg** correspond à un argument passé au **thread** pour l'exécution de la fonction (**start_routine**) (peut valoir NULL si aucun argument est passé en paramètre). La primitive renvoie la valeur 0 en cas de succès, et une valeur négative sinon. Pour utiliser les thread, il conviendra de placer un `<pthread.h>` au début du programme.

- `pthread_exit(void *ret)` met fin au *thread* qui l'exécute, en retournant la valeur `*ret`.
- `pthread_join(pthread_t thread, void **retour)` : permet à un thread d'obtenir la valeur de retour envoyée par un autre thread via la fonction « `pthread_exit()` ». Le paramètre *thread* correspond à l'identifiant du *thread* attendu, et `**retour` contiendra la valeur retournée par l'autre thread. Important : l'exécution du processus qui effectue un appel à la fonction « `pthread_join()` » est suspendue jusqu'à ce que le *thread* attendu se soit achevé.
- Pour créer un mutex, il suffit de créer une variable de type `pthread_mutex_t`, et de l'initialiser avec la constante `PTHREAD_MUTEX_INITIALIZER`. Exemple :

```
pthread_mutex_t pMutex = PTHREAD_MUTEX_INITIALIZER;
```
- `pthread_mutex_destroy(pthread_mutex_t *pMutex)` : permet de détruire proprement le mutex *pMutex*.
- `int pthread_mutex_lock(pthread_mutex_t *pMutex)` : verrouille le mutex *pMutex*. Retourne 0 en cas de succès.
- `int pthread_mutex_unlock(pthread_mutex_t *pMutex)` : déverrouille le mutex *pMutex*. Retourne 0 en cas de succès.
- Pour créer une variable condition, il suffit de créer une variable de type `pthread_cond_t` et de l'initialiser avec la constante `PTHREAD_COND_INITIALIZER`. Exemple :

```
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
```
- `pthread_cond_wait(pthread_cond_t *condition, pthread_mutex_t *mutex)` : permet à un thread de se mettre en attente sur la variable condition *condition*. Le mutex *mutex* devra être verrouillé avant l'appel de cette fonction, et libéré juste après.
- `pthread_cond_signal(pthread_cond_t *condition)` : signale que la condition *condition* est remplie. L'appel à cette fonction doit être protégé par le même mutex que celui protégeant l'appel de la fonction `pthread_cond_wait()`.
- `pthread_cond_destroy(pthread_cond_t *condition)` : détruit la variable condition *condition*.