

Durée : 2 heures

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103 – NANCY/METZ

INTERVENANT : E. DESVIGNE

SESSION DE RATTRAPAGE – 13 SEPTEMBRE 2011

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

Documents autorisés : NON

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

Le jeu de la patate chaude peut se résumer ainsi :

- soit « A » un arbitre ;
- soient 4 joueurs « J0 », « J1 », « J2 », et « J3 » qui ne se connaissent pas au départ. À noter qu'initialement, le score de chaque joueur vaut 0 ;
- au début de la partie, l'arbitre donne l'identité des autres joueurs à chaque joueur ;
- puis l'arbitre désigne un joueur au hasard et lui envoie la patate chaude ;
- dès qu'un joueur reçoit la patate, il incrémente son score de 1. Si le score est inférieur à 1000, il tire au hasard un n° de joueur (il ne doit pas se choisir lui-même) et lui envoie la patate chaude. Sinon, si le score est égal à 1000, il envoie la patate à l'arbitre ;
- quand l'arbitre reçoit finalement la patate (devenue froide), il demande à chaque joueur d'afficher son score, et la partie s'arrête.

Question 1.a) [1 point] : Dans un système d'exploitation de type Unix/Linux, qu'est-ce qu'un processus ?

Question 1.b) [1 point] : Quelle est la différence entre un pipe anonyme et un pipe nommé ?

Question 1.c) [6 points] : Écrire en C (ou éventuellement en pseudo-code) le jeu de la patate chaude, selon les indications suivantes :

- les joueurs « J0 », « J1 », « J2 », et « J3 » sont des processus fils de l'arbitre « A » ;
- en début de partie, « A » communique le PID des autres joueurs via un pipe anonyme ;
- le tirage aléatoire d'un joueur se fait en utilisant la fonction `int num_joueur_aleat()` qui retourne un numéro aléatoire compris entre 0 et 3. À titre d'exemple, voici ce que pourrait être cette fonction :

```
#include <stdlib.h>

int num_joueur_aleat()
{
    long int li;
    li=lrnd48()%4;
    return((int)li);
}
```

- l'envoi de la patate chaude est simulé par l'envoi d'un signal utilisateur `SIGUSR1` (pour les 4 joueurs comme pour le retour de la patate chaude à l'arbitre en fin de partie) ;
- en fin de partie, l'arbitre demande à chaque joueur d'afficher son score (ce qui sous-entend la fin de la partie) par l'envoi d'un signal `SIGUSR2`.

* * * * *

Exercice n°2 : mémoire, segmentation, pagination (3 questions pour un total de 6 points)

Question 2.a) [1 point] : Quel mécanisme de traduction d'adresse permet la mise en place de la mémoire virtuelle (appelé aussi *swap* en anglais) ?

Question 2.b) [2 points] : Il n'est pas rare de trouver, dans les ordinateurs modernes, des bus d'adresses de 32 ou 64 bits, et plusieurs Go de RAM physique. Dans ce cas, il devient impossible pour la MMU d'un CPU de charger la totalité de la table des pages en mémoire. Quelle solution les architectes système et les fondeurs de CPU ont mis au point pour pallier à ce problème ? Faire un schéma illustrant la résolution d'une adresse logique en adresse physique dans cette solution.

Question 2.c) [3 points] : Rappelez les principes de l'algorithme de remplacement des pages « LRU ». Sachant qu'un processus veut accéder aux pages ci-dessous sur un ordinateur équipé de 3 cases (numérotées de 0 à 2) et de 8 pages (numérotées de 0 à 7), faites un schéma qui indique, à chaque accès à une page, l'état de la table des cases, s'il y a un défaut de page, et dans ce cas, quelle est la page victime qui sera choisie pour être stockée sur disque dur :

5, 6, 7, 0, 6, 1, 6, 2, 0, 1, 5, 1, 0, 7, 0, 6, 7, 5, 6, 0.

Donner l'état de la table des pages à la fin de ces accès.

* * * * *

Exercice n°3 : les threads (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 3.a) [1,5 point] : Qu'est-ce qu'un thread ? Quel est l'avantage d'un thread par rapport à un processus ?

Question 3.b) [1,5 point] : Quels sont les deux types de thread ? Pour chacun d'eux, précisez les avantages/inconvénients.

Question 3.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.

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 remplis 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 « `#include <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*.