CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Durée : 2 heures

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103 – NANCY/METZ

EXAMEN – 28 JUIN 2012

Année 2011 – 2012, deuxième semestre
Coefficient : 2/3

Correcteur : Emmanuel DESVIGNE

Documents autorisés : NON

Exercice n°1 : processus, threads *(5 questions pour un total de 7 points)*

Question 1.a) [*1 point*] : Dans un système d'exploitation multiprogrammé de type Unix/Linux, qu'est-ce qu'un « *processus* » ? Quel est le nom de la structure dans laquelle le système d'exploitation stocke les propriétés d'un processus ? Quels sont les états possibles d'un processus ?

Question 1.b) [*1 point*] : Qu'est-ce qu'un « *processus léger* » (appelé « *thread* » dans la littérature anglaise) ? Quels sont les deux types de thread ? Quels sont leur(s) avantage(s)/inconvénient(s) ?

Question 1.c) [*1 point*] : Comment appelle-t-on l'ensemble de codes/programmes d'un système d'exploitation multiprogrammé qui gère l'affectation du CPU aux différents programmes qui ont besoin de faire des calculs ? Citez deux algorithmes qui peuvent être utilisés dans ces programmes, et fournissez-en le principe.

Question 1.d) [*2 points*] : Soit le code C suivant :

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

int main()
{
        int     i,j;
        for (i=1; i <= 3; i++)
        {
                j = fork();
                printf("i=%d, fork()=%d\n", i, j);
        }
        sleep(1);
        exit(0);
}
```

Considérant que la primitive `sleep(n)` permet de bloquer un processus pendant `n` secondes, à l'exécution du programme ci-dessus :

- combien de fois s'affichera une ligne du style : `i=3, fork()=`*`0/ou un nombre`* ?
- combien de fois s'affichera une ligne du style : `i=`*`1/ou 2/ou 3`*`, fork()=`*`0`* ?
- et au final, combien de fois verra-t-on s'afficher une ligne du style : `i=`*`xx`*`, fork()=`*`yy`* (quelle que soit la valeur de x et de y) ?

Remarque : justifier l'ensemble de vos réponses.

Question 1.e) [*2 points*] : Modifiez le code pour que le processus père ne lance que 3 processus fils. Chaque processus fils ne doit pas lui-même lancer de nouveau processus, mais doit simuler un travail en effectuant un « `sleep(1);` ». Le père doit se terminer uniquement une fois que les trois processus fils ont fini leur travail, et en libérant correctement toutes les ressources.

* * * * * *

Exercice n°2 : annuaire téléphonique *(3 questions pour un total de 7 points)*

Vous travaillez dans le service informatique d'une société qui souhaite mettre en place un petit annuaire téléphonique de ses employés. Cet annuaire, stocké dans un fichier unique sur un serveur, sera accessible par plusieurs programmes : un serveur intranet pour proposer un service de type « pages blanches », le logiciel de paie, un programme de mise à jour (ajout/suppression/correction de données dans l'annuaire), etc.

Chaque entrée de l'annuaire devra comprendre :

- Le nom usuel de l'employé (63 caractères max.),
- Ses prénoms (63 caractères max.),
- Le numéro de poste téléphonique (5 caractères max.),
- Son numéro de pager (5 caractères max., optionnel),
- Son adresse électronique (127 caractères max., optionnelle).

Cet annuaire pourra contenir un maximum de 256 entrées, et sera stocké dans un fichier sur le disque dur du serveur.

Tous les programmes qui accèdent à cet annuaire (le logiciel de gestion de l'annuaire, le logiciel de paie qui a besoin de connaître le numéro de poste d'un agent connaissant son nom+prénom, le serveur intranet, etc.) sont des processus de ce même serveur. Ils peuvent être lancés à tout moment, et par conséquent, simultanément.

Question 2.a) ***[1 point]*** Résumer quelle est la problématique d'accès au fichier qui stocke l'annuaire ?

Question 2.b) ***[2,5 points]*** Quelles solutions techniques pourriez-vous proposer pour répondre à cette problématique ? Argumentez les avantages-inconvénients de chaque solution.

Question 2.c) ***[3,5 points]*** Parmi toutes ces solutions, vous avez fait le choix d'une gestion non centralisée du problème, en utilisant les sémaphores (fonctions « `Init(semaphore, val0)` », « `P(semaphore)` », et « `V(semaphore)` »). Proposez le squelette (en C ou en pseudo code) des fonctions :

- « `annuaire_inverse()` » du serveur Intranet qui permet de savoir l'identité de la personne connaissant son numéro de poste téléphonique,
- « `ajouter_une_identite()` » du logiciel de gestion de l'annuaire, qui permet d'ajouter (s'il reste de la place) une entrée dans l'annuaire, en indiquant le nom + prénoms + tel + pager + adresse email de la nouvelle personne à renseigner ;
- « `quel_est_le_numero()` » du logiciel de paie, qui permet de connaître le numéro de téléphone d'un agent, connaissant son nom + prénoms ;
- et « `effacer_une_entree()` » du logiciel de gestion de l'annuaire, dont le rôle est de supprimer une entrée de l'annuaire, connaissant son nom + prénoms.

Les squelettes de ces fonctions ne doivent contenir que le code lié aux problématiques de programmation système/résolution des contraintes liées à la multi-programmation décrites dans la réponse à la question 2.*a)*. Le candidat n'a pas à programmer la gestion du stockage et la recherche des données dans le fichier.

* * * * * *

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

Question 3.a) [*1 point*] : Quel mécanisme de traduction d'adresse permet la mise en place de la mémoire virtuelle (appelée aussi « *swap* » en anglais) ?

Quel module du CPU assure la gestion de cette traduction d'adresse ?

Afin de réduire le nombre d'entrées/sorties sur le disque en cas d'utilisation du mécanisme de mémoire virtuelle, ce module intègre une heuristique basée sur l'utilisation d'un bit « D » (*dirty* en Anglais, souvent traduit par *modifié* dans les livres en français). En quoi consiste cette heuristique (autrement dit, à quoi sert ce bit « D ») ?

Question 3.b) [*1 point*] : Faire un schéma indiquant le mode de fonctionnement du mécanisme de traduction d'adresse logique en adresse physique que vous avez cité dans la question 3.a.

Question 3.c) [*4 points*] : Rappelez les principes des algorithmes de remplacement des pages :

- « **LRU** »,
- et « **algorithme de la seconde chance** ».

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), pour chacun des deux algorithmes « LRU » et « seconde chance », faites un tableau 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 :

3, 4, 5, 6, 4, 7, 4, 0, 6, 7, 3, 7, 6, 5, 6, 4, 5, 3, 4, 6, 5, 4.

Donner l'état de la table des pages à la fin de ces accès, en précisant la valeur du bit V (qui vaut 1 si la page est swapée sur le disque, et 0 sinon).

Comparant les résultats du déroulement des deux algorithmes dans cet exemple particulier, que pouvez-vous conclure ?

* * * * * *

Annexe : les appels systèmes utiles (extrais 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 talbe 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 noumbre négatif en cas d'erreur.
- `int getpid()` : permet à un processus de connaître sont 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 énnoncés 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 msqid, const void *msgp, size_t msgsz, long msgtyp, int msgflg)` : reste bloqué sur la file ayant comme identifiant **msqid** 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 énnoncés ici, sem**flg** 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 blocant.

  En cas de réussite, `semop()` renvoit 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 Ia 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*** cotiendra 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***.