CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Lundi 16/06/2008
2 heures

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103

EXAMEN
Année 2007 – 2008, deuxième semestre
Coefficient : 2/3

Documents autorisés : NON

Cnam
CONSERVATOIRE NATIONAL
DES ARTS ET METIERS

Exercice n°1 : simulateur de vol *(4 questions pour un total de 8 pts)*

Une équipe de programmeurs a pour mission de développer un nouveau jeu de simulation de pilotage d'avion sur ordinateur sous Unix/Linux. De façon tout à fait classique, le joueur commence le jeu avec 3 vies. Il perd une vie dès que son avion est détruit, et il peut en gagner une s'il arrive à passer au travers d'un « *objet bonus* ».

Pour ce, les programmeurs décident que le programme principal doit créer un processus fils, dont le rôle sera de toujours connaître la position de l'avion par rapport aux éléments du décor, afin de détecter toute collision (avec le sol, avec un immeuble, avec un avion adverse, etc.).

Dès que ce processus fils détecte une collision mortelle, il envoie un signal « `SIG_USR1` » au processus père (qui est le programme principal). A chaque réception de ce signal, le processus père exécutera une fonction `detection_collision()`, dont le rôle sera :

- d'appeler une fonction `affiche_explosion()` ce qui aura pour effet d'afficher les éléments graphiques correspondant à l'explosion de l'avion, et de repositionner l'avion au début de l'action (où il ne risque rien) ;
- puis d'appeler une fonction `change_nb_vies(-1)`, ce qui aura comme effet de décrémenter une variable « `nb_vies` », qui est une variable entière globale du programme principal, qui contient le nombre de vies restant au joueur. La valeur retournée par cette fonction « `change_nb_vies()` » est le nombre de vies qu'il reste au joueur. Si ce nombre est égal à zéro, il faut alors appeler la fonction `game_over()`. Sinon, le jeu continue, jusqu'à ce que le signal `SIG_USR1` soit reçu suffisamment de fois pour que le nombre de vies soit épuisé.

Mais si ce même processus fils détecte une collision avec un « *objet bonus* », il enverra cette fois-ci un signal `SIG_USR2` au processus père. A la réception du signal `SIG_USR2`, celui-ci exécutera une fonction `detection_bonus()`, qui ressemble à la fonction `detection_collision()`, sauf que :

- la fonction qui gère l'affichage de l'événement s'appelle `affiche_bonus()`,
- et la fonction `change_nb_vies()` est appelée avec la valeur « `1` » comme paramètre, afin d'incrémenter la variable globale `nb_vies`. Il n'y a évidemment pas à tester si le nombre de vie restantes est nul. Le jeu continue dans tous les cas.

Sans surprise, pour armer ce mécanisme, nous trouvons dans les première lignes de la fonction main() du programme principal les deux lignes suivante :

```
signal(SIG_USR1, detection_collision);
signal(SIG_USR2, detection_bonus);
```

Question 1.a) [*1 point*] : Écrire la fonction « `detection_collision()` ».

Question 1.b) [*2 points*] : En supposant que l'avion soit lancé à pleine vitesse, qu'il percute un immeuble, puis rebondit immédiatement sur un pilonne. Le processus fils détecte bien ces deux collisions. Le jeu réagira-t-il toujours de la même façon (justifiez votre réponse) ? Le joueur pourra-t-il perdre deux vies durant cette courte action ?

Question 1.c) [*1 point*] : Dans le programme principal, une multitude de fonctions exécutées dans différents threads ont besoin de savoir régulièrement combien il reste de vies au joueur (pour l'affichage, pour l'intelligence artificielle qui évalue la probabilité de faire apparaître un objet bonus, etc). Ces routines font alors appel à une fonction « `lit_nb_vies()` », qui retourne le nombre de vies restantes.

Quel est le risque lié aux fonctions `change_nb_vies()` et `lit_nb_vies()` ? Indiquez le nom de la solution la plus efficace (vue en cours) pour pallier à ce risque.

Question 1.d) [*4 points*] : Programmez (en C ou en pseudo-code faisant appel aux appels système Linux) ces deux fonctions `change_nb_vies()` et `lit_nb_vies()`. **Remarque** : pour simplifier, le candidat n'a pas à traiter les codes retour (indiquant des erreurs) des appels système.

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

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 : IPC/MSQ *(4 questions pour un total de 6 pts)*

Contenu du fichier « `mon_msgbuf.h` » :

```
#ifndef     __MON_MSGBUF__
#define     __MON_MSGBUF__

#define cle 0x00654321

struct mon_msgbuf
{
      long  mtype;
      float valeur;
};

#endif      /* __MON_MSGBUF__ */
```

Contenu du fichier « *programmeA.c* » :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include <errno.h>

#include "mon_msgbuf.h"

int main()
{
      int                 code_err_main;
      int                 reponse;
      float               valeur_saisie;
      int                 msqid;
      struct mon_msgbuf   msg;




      msqid = msgget(cle, IPC_CREAT|S_IRUSR|S_IWUSR);
```

```
	if (msqid < 0)
	{
		perror("Erreur à la création du MSQ\n");
		exit(-1);
	}
	do
	{
		printf("Veuillez taper : \n");
		printf("  1 : pour faire un retrait\n");
		printf("  2 : pour faire un dépôt\n");
		printf("  3 : pour quitter tous les programmes\n");
		printf("Faites votre choix ? ");
		fscanf(stdin, "%d", &reponse);
		switch (reponse)
		{
			case 1 :
				printf("Valeur du retrait ? ");
				fscanf(stdin, "%f", &valeur_saisie);
				break;
			case 2 :
				printf("Valeur du dépôt ? ");
				fscanf(stdin, "%f", &valeur_saisie);
				break;
			case 3 :
				printf("Fin normale du programme.\n");
				valeur_saisie = 0.0;
				code_err_main = 0;
				break;
			default :
				/* l'utilisateur n'a saisi ni 1, ni 2, ni 3 */
				printf("Vous avez saisi une valeur erronée.\n");
				printf("Fin anormale de tous les programmes.\n");
				valeur_saisie = -1.0;
				code_err_main = -1;
				reponse = 3;
		}
		msg.mtype = reponse;
		msg.valeur = valeur_saisie;
		if (msgsnd(msqid, (void *)&msg, sizeof(msg.valeur), 0) < 0)
		{
			printf("Erreur lors d'un msgsnd().\n");
			printf("Fin anormale du programme.\n");
			exit(-1);
		}
	} while (reponse != 3);
	/* Ce programme ne détruit pas le MSQ */
	return(code_err_main);
}
```

Question 3.a) [*1 point*] : Expliquez brièvement ce qu'est un « *message queue* » ou « *MSQ* ». Comment sont-il implémentés sous Linux ?

Question 3.b) [*1 point*] : Sous Linux, en dehors de tout mécanisme de cluster ou de virtualisation, les MSQ peuvent ils être utilisés entre deux processus situés sur deux machines différentes ? Entre deux processus situés sur deux CPU différents au sein d'un même ordinateur ?

Question 3.c) [*2 points*] : Complétez le code du programme monotâche « *programmeB.c* » ci-dessous, qui traite les demandes de « *programmeA.c* » :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include <errno.h>

#include "mon_msgbuf.h"

void retrait(float f)
{
      /* code qui traite un retrait de f € */
      bla bla bla...
}

void depot(float f)
{
      /* code qui traite un dépôt de f € */
      bla bla bla...
}

int main()
{
      int                 code_err_main;
      int                 msqid;
      struct mon_msgbuf   msg;
      /* Création du MSQ */
      msqid = msgget(cle, IPC_CREAT|S_IRUSR|S_IWUSR);
      if (msqid < 0)
      {
            perror("Erreur à la création du MSQ\n");
            exit(-1);
      }
```

Votre code ici, faisant appel à la fonction retrait() ou à la fonction depot() selon le message.

```
      return(code_err_main);
}
```

Question 3.d) [*2 points*] : Sans donner les détails du code, expliquez le principe à mettre en oeuvre pour rendre votre programme de la question 3.c) multitâche, une routine traitant les retraits, une autre traitant les dépôts ?

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**.
- `pid_t wait(int *status)` : attend la fin d'un enfant. Si **status** est non NULL, met dans ***status** le code retourné par cet enfant.
- `int kill(pid_t pid, int sig)` : envoie le signal **sig** au processus numéro **pid**.
- `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`**.