Master1 – POSIX EXAMEN Décembre 2007	- Durée : 3 heures - Toute documentation autorisée - Barème indicatif *Olivier Marin*

1. EXERCICE : SOCKETS

On cherche à obtenir une application dans laquelle chaque serveur observe l'activité de tous les serveurs répartis sur le réseau local pour ce faire, on veut écrire un programme comportant deux threads : un thread qui écoute tout ce qui se passe sur son port 9999, et un autre qui diffuse toutes les 3 secondes un message "`I AM ALIVE`" en mode **broadcast**.

On fournit ci-dessous le squelette d'un tel programme :

```
/**** exo-sockets ***/

#define _POSIX_SOURCE 1

/**** ICI DES TAS DE #INCLUDE ****/

#define PORTSERV 9999
#define TAILMSG 80
#define h_addr  h_addr_list[0]
#define BROADCAST_IP "192.168.0.255"
char message[TAILMSG];
int listen_sock, send_sock;

void prep_sockets() {
}

void *listen_heartbeat() {
}

void *send_heartbeat() {
}

int main(int argc, char *argv[]) {
    struct sockaddr_in sin;  /* Nom de la socket du serveur */
    pthread_t tid_listener, tid_sender;

    prep_sockets();

      /* remplir le nom */
    bzero((char *)&sin,sizeof(sin));
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    sin.sin_port = htons(PORTSERV);
    sin.sin_family = AF_INET;

      /* nommage */
    if (bind(listen_sock,(struct sockaddr *)&sin,sizeof(sin)) < 0) {
        perror("bind");
        exit(2);
    }
```

```
    if (pthread_create(&tid_listener, NULL, listen_heartbeat, NULL) != 0)
        perror("pthread_create L");
    if (pthread_create(&tid_sender, NULL, send_heartbeat, NULL) != 0)
        perror("pthread_create S");

    pause();
    return (0);
}
```

1.1. (1 pt)

Complétez le code de la procédure `prep_sockets`, qui crée les sockets d'écoute et d'envoi. Vous penserez à déclarer le mode broadcast lorsque cela s'avère nécessaire.

1.2 (2 pts)

Complétez le code de la procédure `listen_heartbeat`, qui boucle en attente de messages "`I AM ALIVE`" et affiche l'adresse de l'émetteur à chaque réception.

1.3 (3 pts)

Complétez le code de la procédure `send_heartbeat`, qui diffuse un message "`I AM ALIVE`" toutes les 3 secondes sur l'adresse `BROADCAST_IP`.

2. PROBLEME : IPCS & THREADS

On travaille maintenant sur une application qui fait correspondre de manière tout à fait classique des lecteurs et des écrivains, mais **sans utiliser de file de messages**. Un écrivain accède à un segment de mémoire partagée et y dépose un message (son pid) afin que celui-ci soit lu par **tous** les lecteurs. Un message ne peut être écrasé tant que tous les lecteurs n'en ont pas eu connaissance. Le programme est composé d'un processus principal qui démarre `NB_LECTEURS` lecteurs et `NB_ECRIVAINS` ecrivains, puis se suspend en attente d'un signal d'interruption (`SIGINT`). A la réception de ce signal, le processus principal attend la fin de tous ses fils puis détruit les IPCs créés.

Le code (quasi-complet) de ce programme vous est fourni en annexe.

2.1 (1,5 pts)

Complétez le code manquant dans les procédures `init_sigs`, `wait_for_signal`, et `close_app`.

A l'exécution, on s'aperçoit que ce code comporte des erreurs manifestes.

Tout d'abord il arrive que plusieurs écrivains parviennent à modifier successivement le contenu de la mémoire partagée. Voici un exemple d'affichage obtenu avec 2 écrivains et 3 lecteurs.

```
[marin@blake Exam2007]$ gcc -o test exo-ipc.c -Wall
[marin@blake Exam2007]$ ./test
8422> Starting as ecrivain
8422> Writing
8423> Starting as ecrivain
8419> Starting as lecteur
8420> Starting as lecteur
8420> msg = 8422
8419> msg = 8422
```

```
8421> Starting as lecteur
8421> msg = 8422
8423> Writing
8422> Writing
8420> msg = 8423
8419> msg = 8423
8421> msg = 8423
8422> Writing
8420> msg = 8423
8419> msg = 8422
8421> msg = 8422
^C8423> ## closing forked process
8421> ## closing forked process
8420> ## closing forked process
8419> ## closing forked process
8422> ## closing forked process
8418> ## closing main
```

2.2 (1,5 pts)

Quelles modifications faut-il apporter pour pallier ce problème particulier (et seulement celui-ci) ? Détaillez le code à ajouter/enlever.

Une autre anomalie qui transparaît à l'exécution est la suivante : un même lecteur peut lire plusieurs fois le même message à la place d'autres lecteurs, comme le montre l'exemple d'affichage ci-dessous.

```
[marin@blake Exam2007]$ ./test
8472> Starting as ecrivain
8473> Starting as ecrivain
8473> Writing
8470> Starting as lecteur
8469> Starting as lecteur
8469> msg = 8473
8469> msg = 8473
8469> msg = 8473
8473> Writing
8471> Starting as lecteur
8469> msg = 8473
8470> msg = 8473
8471> msg = 8473
^C8471> ## closing forked process
8472> ## closing forked process
8473> ## closing forked process
8469> ## closing forked process
8470> ## closing forked process
8468> ## closing main
```

2.3 (3 pts)

Sans reprendre tout le code, détaillez les éléments que vous avez besoin d'ajouter (variables, sémaphores, …), puis donnez les modifications à apporter aux seules procédures `lecteur` et `ecrivain` pour résoudre ce problème particulier.

On veut maintenant reprendre ce programme pour remplacer les processus par des threads. Vous trouverez en Annexe 2 un squelette de programme qui vous permettra de répondre plus facilement aux questions qui suivent.

2.4 (2 pts)

Complétez le code de `start_app`, qui lance l'exécution des NB_LECTEURS (respectivement NB_ECRIVAINS) threads lecteurs (resp. écrivains) et qui initialise la valeur de `cpt`.

2.5 (2 pts)

Complétez le code des procédures `init_sigs` et `wait_for_interruption` de sorte que seul le thread principal traite le signal `SIGINT` en appelant `close_app`.

2.6 (4 pts)

Complétez le code de la procédure ecrivain pour la rendre cohérente avec la procédure lecteur.

ANNEXE 1 : PROGRAMME "EXO-IPC.C"

```
 1:  #define _POSIX_SOURCE 1
 2:
 3:  /**** ICI DES TAS DE #INCLUDE ****/
 4:
 5:  #define NB_LECTEURS 3
 6:  #define NB_ECRIVAINS 2
 7:
 8:  typedef struct shared_space {
 9:      int cpt;
10:      int msg;
11:  } shared;
12:  shared *buf;
13:  int shm_fd;
14:  sem_t *rd_sem, *wr_sem;
15:
16:  int main_pid;
17:  int pid_lecteurs[NB_LECTEURS];
18:  int pid_ecrivains[NB_ECRIVAINS];
19:
20:  void close_app(int sig) {
21:      int i;
22:      if (getpid() == main_pid) {
23:          /**** A COMPLETER ****/
24:          printf("%d> ## closing main\n", getpid());
25:      } else {
26:          printf("%d> ## closing forked process\n", getpid());
27:          exit(0);
28:      }
29:  }
30:
31:  void init_sigs(sigset_t *mask) {
32:      /**** A COMPLETER ****/
33:  }
34:
35:  void wait_for_signal(sigset_t *mask) {
36:      /**** A COMPLETER ****/
37:  }
38:
39:  void init_ipc() {
40:      rd_sem = sem_open("/rd_sem:0", O_CREAT|O_RDWR, 0600, 0);
41:      wr_sem = sem_open("/wr_sem:0", O_CREAT|O_RDWR, 0600, 1);
42:      shm_fd = shm_open("/shm:0", O_CREAT|O_RDWR, 0600);
43:      ftruncate(shm_fd, (sizeof(shared)));
44:      if ((buf=(shared*)mmap(NULL,(sizeof(shared)), PROT_READ|PROT_WRITE
45:                                  MAP_SHARED, shm_fd, 0)) == MAP_FAILED) {
46:          perror("mmap");
47:          exit(1);
48:      }
49:  }
```

```
50:
51:   void ecrivain() {
52:       int i;
53:       printf("%d> Starting as ecrivain\n", getpid());
54:       while(1) {
55:           sem_wait(wr_sem);
56:           printf("%d> Writing\n", getpid());
57:           buf->cpt = 0;
58:           buf->msg = getpid();
59:           for (i = 0; i < NB_LECTEURS; i++)
60:              sem_post(rd_sem);
61:       }
62:   }
63:
64:   void lecteur(int lid) {
65:       printf("%d> Starting as lecteur\n", getpid());
66:       while(1) {
67:           sem_wait(rd_sem);
68:           printf("%d> msg = %d\n", getpid(), buf->msg);
69:           buf->cpt++;
70:           if (buf->cpt == NB_LECTEURS)
71:              sem_post(wr_sem);
72:       }
73:   }
74:
75:   void start_app() {
76:       int i;
77:       for (i = 0; i < NB_LECTEURS; i++) {
78:           if ((pid_lecteurs[i] = fork()) == 0)
79:              lecteur(i);
80:       }
81:       for (i = 0; i < NB_ECRIVAINS; i++) {
82:           if ((pid_ecrivains[i] = fork()) == 0)
83:              ecrivain();
84:       }
85:   }
86:
87:   int main(int argc, char **argv) {
88:
89:       sigset_t mask;
90:
91:       init_sigs(&mask);
92:       init_ipc();
93:
94:       main_pid = getpid();
95:
96:       start_app();
97:
98:       wait_for_signal(&mask);
99:
100:      return EXIT_SUCCESS;
101:  }
```

ANNEXE 2 : PROGRAMME "EXO-THREADS.C"

```
#define _POSIX_SOURCE 1

/**** ICI DES TAS DE #INCLUDE ****/

#define NB_LECTEURS 5
#define NB_ECRIVAINS 3

int msg;
int cpt;

pthread_mutex_t mutex;
pthread_cond_t cond_read, cond_write;
pthread_t tid_lecteurs[NB_LECTEURS];
pthread_t tid_ecrivains[NB_ECRIVAINS];

void close_app(int sig) {
    printf("Ending process\n");
    exit(0);
}

void init_sigs(sigset_t *mask) {
    /**** A COMPLETER ****/
}

void init_ipc() {
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond_read, NULL);
    pthread_cond_init(&cond_write, NULL);
}

void *ecrivain() {
    /**** A COMPLETER ****/
}

void *lecteur(void* lid) {
    printf("%d> Starting as lecteur %d\n", (int)pthread_self(), (int)lid);
    while(1) {
        pthread_mutex_lock(&mutex);
        cpt++;
        if (cpt == NB_LECTEURS)
               pthread_cond_signal(&cond_write);
        pthread_cond_wait(&cond_read, &mutex);
        pthread_mutex_unlock(&mutex);

        printf("%d> msg = %d\n", (int)pthread_self(), msg);

    }
}

void start_app() {
    /**** A COMPLETER ****/
}

void wait_for_interruption(sigset_t *mask) {
    /**** A COMPLETER ****/
}
```

```c
int main(int argc, char **argv) {
    sigset_t mask;

    init_sigs(&mask);

    init_ipc();

    start_app();

    wait_for_interruption(&mask);

    return EXIT_SUCCESS;
}
```