Université Joseph Fourier, Grenoble
Licence d'Informatique L3

Introduction aux Systèmes et Réseaux

TP n°5 : Synchronisation.

1 Synchronisation par attente active

1.1 Question

Pour implémenter et tester la solution de Peterson pour deux processus (vu en cours et TD), les processus ont besoin de partager des données (ils travaillent sur les mêmes variables). Est-ce possible dans une configuration par défaut ?

1.2 Mémoire partagée entre processus

Le système d'exploitation fournit des fonctions dédiées à la création et à la gestion d'un espace mémoire partagé entre processus. Un processus peut créer un espace mémoire partagé en utilisant la primitive

```
int shmget(key t key, size t size, int shmflg);
```

f Une fois créé, l'espace mémoire partagé doit être attaché au processus pour être utilisé. Ceci se fait avec la primitive

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Voici un exemple où un processus crée un espace mémoire partagé et y écrit des données. Un deuxième processus lit les données.(Les fichiers sont fournis sur Moodle.)

--- shm_server.c ---

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define SHMSZ     27

void main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    /* We'll name our shared memory segment"5678 */
    key = 5678;

    /* Create the segment. */
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /* Now we attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /* Now put some things into the memory for the other process to read. */
    s = shm;

    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = '\0';

    /*
     * Finally, we wait until the other process changes the first character of our memory
     * to '*', indicating that it has read what we put there.
     */
    while (*shm != '*')
        sleep(1);

    exit(0);
}
```

--- shm_client.c ---

```
/*
 * shm-client - client program to demonstrate shared memory.
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define SHMSZ     27

void main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We need to get the segment named "5678", created by the server.
     */
    key = 5678;

    /* Locate the segment. */
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /* Now we attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /* Now read what the server put in the memory.*/
    for (s = shm; *s != '\0'; s++)
        putchar(*s);
    putchar('\n');

    /*
     * Finally, change the first character of the
     * segment to '*', indicating we have read
     * the segment.
     */
    *shm = '*';

    exit(0);
}
```

1.3 Question

Lire, comprendre et tester l'exemple fourni.

1.4 Question

En utilisant les primitives de mémoire partagée, implémenter la solution de synshronisation de Peterson pour deux processus.

2 Masquage des signaux, attente et synchronisation

2.1 Masquage de signaux

Un processus peut bloquer la réception des signaux d'un certain type. Un signal bloqué parvient au processus destinataire, mais ne provoque aucun effet jusqu'à ce qu'il soit débloqué (c'est l'équivalent du masquage des interruptions matérielles).

Pour le masquage, il faut manipuler un tableau de bits qui représente le masque (un bit par signal, 1=bloqué, 0=non bloqué). Le type opaque `sigset_t` représente un tel tableau de bits pour l'ensemble des signaux. L'interface pour la manipulation de ce tableau est donnée ci-après (l'effet est indiqué en commentaire).

```
int sigemptyset(sigset_t *p_ens)            // *p_ens = ensemble vide
int sigfillset(sigset_t *p_ens)             // *p_ens = {1, ..., NSIG-1}
int sigaddset(sigset_t *p_ens, int sig)     // *p_ens = *p_ens U {sig}
int sigdelset(sigset_t *p_ens, int sig)     // *p_ens = *p_ens - {sig}
int sigismember(sigset_t *p_ens, int sig)   // sig appartient à *p_ens?
```

Le masquage proprement dit est réalisé par la primitive :
`int sigprocmask(int op, const sigset_t *p_ens, sigset_t *p_ensAncien);`
qui modifie le masque du processus en fonction de la valeur de `op`.

- `op`=`SIG_SETMASK` : nouveau masque = `*p_ens`
- `op`=`SIG_BLOCK` : nouveau masque = masque courant U `*p_ens`
- `op`=`SIG_UNBLOCK` : nouveau masque = masque courant - `*p_ens`

Si `*p_ensAncien` est différent de `NULL`, alors l'ancienne valeur du masque est stockée dans cette variable.

La primitive `int sigpending(sigset_t *p_ens)` écrit dans `*p_ens` la liste des signaux pendants qui sont bloqués.

2.2 Observation du masquage de signaux

Le programme suivant (`mask.c`) place un masque sur 2 signaux (`SIGINT` et `SIGUSR1`), s'envoie ces signaux, imprime la liste des signaux pendants masqués, puis débloque les signaux masqués.

```
void handler_sigint(int sig)
{
    printf("Caught SIGINT\n");
    return;
}

void handler_sigusr1(int sig)
{
    printf("Caught SIGUSR1\n");
    return;
}

int main()
{
    sigset_t s1;
    sigset_t s2;
    int i;

    printf("NB_SIG=%d, SIGRT_ON=%d\n", NB_SIG, SIGRT_ON);

    Signal(SIGINT, handler_sigint);
    Signal(SIGUSR1, handler_sigusr1);

    Sigemptyset(&s1);
    Sigaddset(&s1, SIGINT);
    Sigaddset(&s1, SIGUSR1);
    Sigprocmask(SIG_BLOCK, &s1, NULL);
    printf("Signals SIGINT (%d) and SIGUSR1 (%d) should now be blocked\n",
           SIGINT, SIGUSR1);

    printf("Sending SIGINT to myself\n");
    Kill(getpid(), SIGINT);
    printf("Sending SIGUSR1 to myself\n");
    Kill(getpid(), SIGUSR1);

    Sigemptyset(&s2);
    sigpending(&s2);
    for (i=1; i < NB_SIG; i++) {
        if (Sigismember(&s2, i)) {
            printf("Signal %d is currently pending\n", i);
        }
    }

    printf("Unblocking SIGINT and SIGUSR1\n");
    Sigprocmask(SIG_UNBLOCK, &s1, NULL);
    sleep(1);

    exit(0);
}
```

2.3 Question

Ecrire un programme avec les caractéristiques suivantes :
— création d'un processus fils (le père se bloque ensuite en attente de la terminaison du fils) ;
— le processus fils est protégé contre le signal `SIGINT` pendant la première partie de son travail, qui consiste à incrémenter un compteur (ce processus doit être protégé contre le signal `SIGINT` dès qu'il commence à s'exécuter) ;
— le processus fils démasque ensuite le signal `SIGINT` et se bloque en attente de celui-ci via un appel à `pause`.
— le traitant associé à `SIGINT` affiche le message "*received SIGINT*" et se termine.
— après avoir été débloqué par le traitant de `SIGINT`, le processus fils affiche le message "*Bye*" et se termine.

2.4 Question

Le programme précédent peut poser un problème si le signal attendu n'est envoyé qu'une seule fois et à un moment inopportun. Expliquer et mettre en évidence ce problème, en utilisant la commande `kill` (pour envoyer un signal) et, si nécessaire, en insérant une boucle de temporisation dans le programme.

2.5 Question

Afin de résoudre le problème mis en évidence à la question précédente, on propose d'utiliser la primitive `sigsuspend` fournie par le noyau pour effectuer le démasquage et bloquer le processus jusqu'à l'arrivée d'un signal.

Avec l'aide de la documentation [1], expliquer le fonctionnement de cette primitive.

Quelle est sa propriété essentielle qui permet de résoudre le problème évoqué à la question précédente [2] ?

Mettre en œuvre cette modification.

3 Bibliothèque de primitives pour la synchronisation

Voici une implémentation des primitives de synchronisation *tell/wait* utilisées dans le sujet du TD n°4. L'implémentation utilise les mécanismes de signaux (traitants, masquage, démasquage). Elle utilise `SIGUSR1` (respectivement `SIGUSR2`) pour permettre à un processus d'envoyer une notification à son père (resp. fils).

1. Il est conseillé de consulter les documentations suivantes : page de man (`man 3 sigsuspend`) et documentation de la glibc, section 24.8 (`http://www.gnu.org/s/libc/manual/html_node/Waiting-for-a-Signal.html`)

2. Cette propriété n'est pas nécessairement mentionnée explicitement dans la page de manuel (en particulier sous Linux).

--- tell_wait.c ---

```
#include "csapp.h"
#include "tell_wait.h"

volatile sig_atomic_t sigflag_child;
volatile sig_atomic_t sigflag_parent;
sigset_t newmask, oldmask;

void sigusr1_handler(int signo)
{
    sigflag_child = 1;
}

void sigusr2_handler(int signo)
{
    sigflag_parent = 1;
}

void init_synchro()
{
// Register signal handlers
// They will be inherited by the child
    Signal(SIGUSR1, sigusr1_handler);
    Signal(SIGUSR2, sigusr2_handler);

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);

    // Add SIGUSR1 and SIGUSR2 to the list of blocked signals
    // and save the previous signal mask
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) {
        unix_error("SIG_BLOCK error");
    }
}

void tell_parent()
{
    kill(getppid(), SIGUSR1);
}

void wait_parent()
{
    // Change the signal mask (SIGUSR1 and SIGUSR2 unblocked)
    // The while loop is required to make sure that
    // the process was awakened by the SIGUSR2 signal sent by its parent
    while (sigflag_parent == 0) {
        sigsuspend(&oldmask);
    }

    // When we reach this point, the signal mask is in the same state
```

```
    // as it was before calling sigsuspend
    // (oldmask + SIGUSR1 and SIGUSR2 blocked)

    // Reset signal flag
    sigflag_parent = 0;
}

void tell_child(pid_t p)
{
    kill(p, SIGUSR2);
}

void wait_child()
{
    // The while loop below is required to make sure that
    // the process was awakened by the SIGUSR1 signal sent by its child
    while (sigflag_child == 0) {
        sigsuspend(&oldmask);
    }

    // When we reach this point, the signal mask is in the same state
    // as it was before calling sigsuspend
    // (oldmask + SIGUSR1 and SIGUSR2 blocked)

    // Reset signal flag
    sigflag_child = 0;
}
```

3.1 Question

Lire et comprendre la solution fournie.

3.2 Question

Mettre en œuvre plusieurs programmes pour tester le bon fonctionnement des primitives de synchronisation.