IG2I
Institut de Génie Informatique et Industriel

LA1 - Année 2016-2017

TP n°4 - Mémoire partagée et sémaphores nommés

Ce TP aborde les mécanismes de mémoire partagée, très utiles lorsqu'il est nécessaire de faire communiquer des processus appartenant à des utilisateurs différents. L'exercice de synthèse permettra d'implémenter la solution du problème classique des « rédacteurs-lecteurs » dans les cas 1-1 ; 1-N et N-M.

Ce TP fera l'objet d'un compte-rendu contenant :

Les réponses aux diverses questions présentes dans cet énoncé,
Les codes sources commentés des programmes réalisés,
Les jeux d'essais obtenus,
Une conclusion sur le travail réalisé, les difficultés rencontrées, les connaissances acquises ou restant à acquérir, …

Il sera tenu compte de la qualité de vos pratiques de production de code :
Lisibilité du code : emploi de commentaires, indentation, constantes symboliques
Automatisation : Fichiers makefile avec macros …
Robustesse : Vérification des valeurs de retours des appels systèmes

1 Mémoire partagée

Comme on l'a vu précédemment, les threads d'un même processus disposent d'une mémoire commune qu'ils manipulent par l'intermédiaire de variables globales. Ce partage mémoire n'est plus assuré pour les processus, même entre un processus fils et son père. Les solutions qu'il est alors possible de mettre en œuvre reposent soit sur des entrées-sorties par fichiers, soit sur des mécanismes de mémoire partagée qui ont pour effet de rendre visible le contenu d'un fichier comme s'il s'agissait d'un segment mémoire du processus.

1.1 Utilisation de fichiers

1. Quels sont les 4 appels système[1] permettant de manipuler des fichiers (textes ou binaires) sous Unix : ouvrir, lire, écrire et atteindre un emplacement précis dans le fichier. Donner un exemple d'utilisation de chacune d'entre elles.

[1] On parle ici de la famille de fonctions « open »… et pas « fopen »

2. Dans le cas d'un partage de données entre deux processus (père-fils) à l'aide d'un fichier, que devient le descripteur du fichier ouvert lorsque que le père exécute l'appel système fork() ?
3. Compléter le fichier `PRS_tp4_1.c`, de façon à ce que le processus fils ajoute à plusieurs reprises, au fichier partagé `PRS_tp4_1.dat` la valeur d'une matrice 2×2. Ce fichier est ensuite lu composante par composante, par le processus père.

1.2 Utilisation de mémoire partagée

1. Quelles sont les fonctions permettant de manipuler la mémoire partagée sous Unix. Préciser également la séquence d'utilisation de ces fonctions.
2. Où sont enregistrés les pseudo-fichiers de mémoire partagée ?
3. De quelle manière les droits doivent-ils être positionnés pour permettre un partage de cette mémoire pour des utilisateurs différents ? Fournir un exemple.
4. Quel est le rôle de la fonction msync ?
5. Comment garantir l'accès exclusif à une zone mémoire partagée entre plusieurs processus d'utilisateurs différents ?

2 Le problème des rédacteurs-lecteurs

Le problème des rédacteurs-lecteurs[2] consiste à mettre en œuvre un ou plusieurs processus dénommés « rédacteurs » qui produisent des données dans un tampon de lecture/écriture de taille borné. Ces données sont à la disposition d'un ou plusieurs processus « lecteurs » qui vont lire les données à leur rythme suivant leurs besoins. Lorsqu'une donnée a été lue par tous les lecteurs, l'emplacement qu'elle occupait dans le tampon est libéré et peut de nouveau être utilisé par le(s) rédacteur(s).

2.1 Structure du tampon

Le tampon qui sera créé en mémoire partagée, peut être défini comme une structure à trois composantes :

```
typedef struct {

    int  next_index;     /* indice de la prochaine case à remplir */

    int  next_value;     /* prochaine valeur à écrire             */
```

[2] A ne pas confondre avec le problème des « producteurs-consommateurs » également présent dans la littérature : la notion de « consommation » induit la disparition des ressources, ce qui n'est pas le cas lors d'une « lecture ».

```
    int  buff[FIFO_SIZE];
} buffer_t;
```

La première composante contient la position du pointeur d'écriture, c.-à-d. l'indice de la prochaine case où écrire. Les données produites sont de simples entiers rangés séquentiellement dans la composante `buff`, gérée comme un buffer circulaire : la case d'indice 0 sera la suivante à écrire, lorsqu'on vient d'écrire dans la dernière case de `buff`. La prochaine valeur à écrire est rangée dans la seconde composante de la structure de tampon (`next_value`).

Les opérations de modification du tampon sont exclusives : plusieurs rédacteurs ne peuvent modifier simultanément une ou plusieurs composantes du tampon. Par contre, plusieurs lecteurs peuvent accéder au tampon simultanément en lecture.

2.2 Cas 1 rédacteur – 1 lecteur

Deux programmes doivent être réalisés :

le premier programme implémente le rédacteur

l'autre, le lecteur

1. Modéliser ce cas à l'aide d'un réseau de Petri partiel faisant essentiellement apparaître :
   - L'opération `ecrire(i)` (écrire dans la case i du tampon) ;
   - L'opération `lire(i)` (lire le contenu de la case i du tampon) ;
   - ainsi que le mécanisme à base de sémaphore à compte, garantissant que le lecteur ne lise pas une case vide, et que le rédacteur n'écrive pas dans une case dont le contenu n'a pas encore été lu par tous les lecteurs.
1. Écrire dans le fichier `lect_red.h`, les déclarations communes au rédacteur et au lecteur.
2. Écrire le programme ***`redacteur`***, qui prend en charge la création et l'initialisation de l'espace mémoire partagé ainsi que la création et l'initialisation des sémaphores nommés. Il produit ensuite une série de valeurs successives de 1 jusqu'à une valeur finale.
3. Écrire le programme ***`lecteur`***, chargé de lire les valeurs produites par le rédacteur, dans l'ordre avec lequel elles ont été produites. Ce programme s'arrête dès que la valeur finale générée par le rédacteur a été lue.

2.3 Cas 1 rédacteur – n lecteurs

Un message produit par le rédacteur dans une case du tampon doit être lu par tous les lecteurs. Leur nombre `NB_LECT` est fixé dans le fichier `lect_red.h`.

1. Modéliser le cas d'un rédacteur et de deux lecteurs à l'aide d'un réseau de Petri
2. Apporter les modifications nécessaires au fichier `lect_red.h`
3. Modifier en conséquence le fichier `redacteur.c`

4. Modifier également le fichier `lecteur.c` de façon à pouvoir récupérer sur la ligne de commande du numéro qui doit être compris entre 1 à `NB_LECT`.

2.4 Cas n Rédacteurs – m lecteurs

Les rédacteurs se partagent les opérations d'écriture dans le tampon ; il faut simplement garantir qu'ils ne vont pas accéder simultanément au tampon, ce qui aurait comme conséquences, soit qu'ils écrivent la même valeur dans des cases différentes, soit qu'ils écrivent des valeurs différentes dans la même case du tampon.

1. Les traitements effectués par les lecteurs n'étant pas impactés par cette nouvelle modification, modéliser le cas de deux rédacteurs se partageant la production de données dans le tampon.
2. Apporter les modifications nécessaires aux fichiers `lect_red.h` et `redacteur.c`.

3 Annexes

3.1 Le fichier initial PRS_tp4_1.c

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/wait.h>

/*    Ajouter ici les #include nécessaires à l'utilisation des
 *    fonctions Unix de manipulation de fichiers
 */
#include <...>
#include <...>

#ifdef NO_DEBUG
#define     CHECK(sts, msg)                     \
      if ( -1 == (sts))                         \
      {                                         \
            perror(msg);                  \
            exit(EXIT_FAILURE);                 \
      }                                         \
      else printf("%s ... Ok\n", msg)
#else
#define     CHECK(sts, msg)                     \
      if ( -1 == (sts))                         \
      {                                         \
            perror(msg);                  \
            exit(EXIT_FAILURE);                 \
      }
#endif

void  processFils(int);

int main (void)
{
      pid_t       pid;
      int         fd;
//    doubleMatrice[2][2] = {{0.}};
      double x;
      int         i, nb;

      /*    Compléter l'opération CHECK ci-dessous de façon à réaliser
       *    l'ouverture du fichier Lse3_tp4.dat
       *    Préciser la signification du mode d'ouverture
       *    Préciser la signification des permissions attribuées
       */
      CHECK(fd = ..., " ...()");
```

```
	pid = fork();
	switch(pid)
	{
		case -1:
			perror("fork");
			exit(EXIT_FAILURE);

		case  0:
			processFils(fd);

		default:	/** suite du processus père   */
			CHECK(waitpid(pid, NULL, 0), "waitpid()");

			/*	Compléter l'opération CHECK ci-dessous de façon à
			 *  revenir au début du fichier
			 */
			CHECK(..., "...()");

			/*	Compléter l'opération CHECK ci-dessous, de façon à
			 *	lire la première valeur réelle enregistrée dans le
			 *	fichier
			 */

			CHECK(nb = ..., "...()");

			for (i = 0; nb > 0 ; i++)
			{
				if (i % 4 == 0)
					printf("\n%4d :\t", i + 1);
				printf("%11.6f\t", x);

				/*	Compléter l'opération CHECK ci-dessous,
				 *	de façon à lire la valeur réelle suivante
			 *	dans le fichier
				 */
				CHECK(nb = ..., "...()");
			}

			printf("\n----------------------------------------"
				"----------------------------------------\n");
	}

	/*	Compléter l'opération CHECK ci-dessous, de façon à réaliser
	 *	la fermeture du fichier
	 */
	CHECK(..., "...()");

	return EXIT_SUCCESS;
}

void  processFils(int fd)
{
	double Mat[2][2] =
		{
```

```
                { 3.141592653589793, 2.7182818284590452},
                { 1.414213562373095, 1.6180339887498948}
        };
    int i;

    for (i = 0; i < 3; i++)
    {
        /*    Compléter l'opération CHECK ci-dessous de façon à
         *    enregistrer le contenu complet de la matrice Mat.
         */
        CHECK(..., "...()");

        /* Exemple de mise à jour de la matrice         */
        Mat[0][0] *= Mat[0][0]; Mat[0][1] *= Mat[0][1];
        Mat[1][0] *= Mat[1][0]; Mat[1][1] *= Mat[1][1];
    }
    exit(EXIT_SUCCESS);
}
```

3.2 Fichier lect_red.h

```
#ifndef _LECT_RED_H_
#define _LECT_RED_H_

/** Noms des fichiers spéciaux associés aux objets du noyau
 *    !!! doivent être modifiés si utilisation d'un serveur commun à plusieurs
étudiants
 */
#define     SEM_RED_NAME"sem_red"
#define     SEM_LECT_NAME     "sem_lect"
#define     MUTEX_NAME        "mutex"
#define     MMAP_NAME         "mapping"

/** Nombre de "cases" du tampon partagé par les processus lecteurs
 *    et rédacteurs
 */
#define     FIFO_SIZE         5

/**   Nombre de processus lecteurs (lecteurs)                    */
#define     NB_LECT           3

/**   Dernière valeur écrite (par l'un des rédacteurs) et lue par
 *    chaque lecteur
 */
#define     VALEUR_FIN      25

/**   Structure du tampon partagé par les rédacteurs et les lecteurs
 *    Note: les deux premières composantes ne sont réellement utiles
 *          que dans le cas où il y a plusieurs rédacteurs
 */
typedef struct {
    int  next_index;     /* indice de la prochaine case à remplir  */
    int  next_value;     /* prochaine valeur à écrire              */
```

```
    int  buff[FIFO_SIZE];/* le buffer                                    */
} buffer_t;


/**   Variante de la macro CHECK définie dans les TP précédents.
 *    Lorsque stat vaut val alors l'opération dont le résultat
 *    est stat, a échoué.
 *    Pour la plupart des appels système Posix, la valeur signalant
 *    l'échec est -1. Elle correspond à NULL lorsque l'appel système
 *    retourne un pointeur.
 */
#ifdef_DEBUG
#define CHECK_IF(stat, val, msg)     \
      if ( (stat) == (val) )         \
      {                              \
            perror(msg);             \
            exit( EXIT_FAILURE );    \
      }
#else
#define CHECK_IF(stat, val, msg)     \
      if ( (stat) == (val) )         \
      {                              \
            perror(msg);             \
            exit( EXIT_FAILURE );    \
    }                                \
    else printf("%s ... OK\n", msg)

#endif
#endif
```