Notes sur la gestion des segments de mémoire partagée

Master 1^{iere} année, Bruno Jacob, IC2

1 Introduction

Ce mécanisme permet à des processus quelconques de partager des zones de données physiques, éventuellement structurées. Les zones partagées par différents processus apparaissent comme des ressources critiques et leur utilisation se fera souvent conjointement avec celle des sémaphores.
L'avantage d'un tel mécanisme de communication par rapport aux files de messages est qu'il n'entraîne aucune recopie de données et est par conséquent beaucoup plus performant.

2 Gestion mémoire des processus

Jusqu'alors, la gestion de la mémoire nous était transparente et nous déléguions cela aux primitives du style `malloc`. Pour partager cette mémoire nous devons nous pencher un peu plus précisément sur son organisation dont voici les grandes lignes.

2.1 Espace d'adressage d'un processus

Le code d'un programme exécutable est "translatable" c'est à dire que toutes les adresses qu'il contient sont "virtuelles" et que l'on effectue une translation pour les charger en mémoire à des adresses physiques quelconques. C'est le module de gestion de la mémoire qui se charge de traduire ces adresses virtuelles en adresses physiques au cours de l'exécution d'un programme. Un processus a donc un espace d'adressage virtuel divisé traditionnellement en 3 régions logiques

- les instructions
- les données
- la pile

La mémoire physique est organisée en pages (unité de mémoire allouable), de taille variable selon les machines. L'espace virtuel est découpé en pages logiques. Une table de pages contient toutes les informations relatives aux pages logiques. L'association entre une adresse virtuelle et une adresse physique se réalise avec cette table de pages qui fait correspondre les pages logiques et les pages physiques.

2.2 Organisation générale de la mémoire

L'espace d'adressage virtuel d'un processus peut être shématisé comme dans la figure Fig. 1 . La zone de données et la pile ont la possibilité de

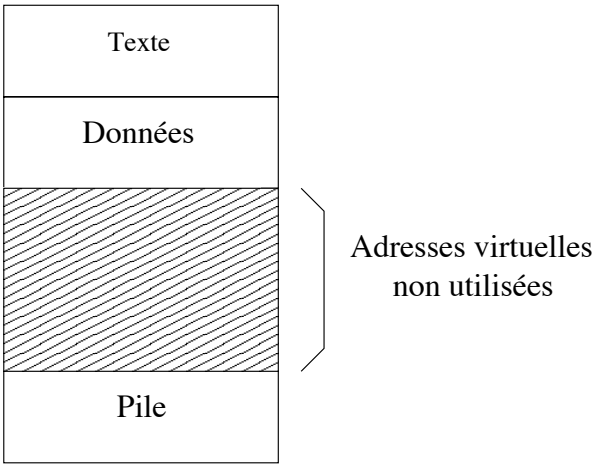

Figure 1 – Adressage virtuel d'un processus

croître :

— la pile par les sauvegardes des paramètres lors de l'appel de fonctions
— les données par l'allocation de variables dynamiques

2.3 Le point de rupture

L'utilisateur ne peut pas manipuler la pile qui est gérée automatiquement par le système, en revanche il peut agir sur la zone des données à travers le point de rupture. Le point de rupture (ou *breakpoint*) est l'adresse virtuelle la plus proche située hors de la zone de données.
En principe, toute adresse située au delà du *breakpoint* est incorrecte. Cependant, comme le système alloue la mémoire par page, il est possible d'accéder à toute une page (déduite de la position du point de rupture). En revanche

toute tentative d'écriture au delà de la limite de cette page provoquera une erreur mémoire matérialisée par le signal `SIGSEGV` dont le taitement par défaut est de planter le processus et d'afficher le célèbre message *"Segmentation fault"*. Le schéma d'organisation de l'espace d'adressage d'un processus peut donc être affiné comme dans la figure FIG. 2 .

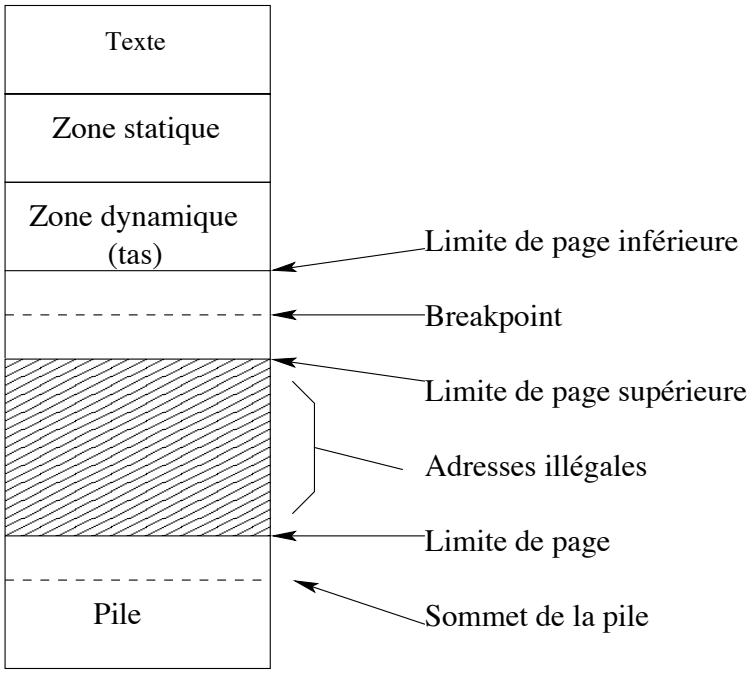

FIGURE 2 – Position du breakpoint dans l'espace d'adressage d'un processus

2.4 Déplacement du point de rupture

L'utilisateur à la possibilité de déplacer le point de rupture par 2 primitives :

```
#include <unistd.h>
int brk(void *p_brk); /* Nouvelle valeur du breakpoint */
```

Positionne le point de rupture à l'adresse `p_brk`. Renvoie

- 0 si OK
- -1 sinon (si l'adresse donnée dépasse la limite imposée par le système par exemple)

```
#include <unistd.h>
void *sbrk(intptr_t inc); /* Deplacement du breakpoint */
```

Ajoute la valeur `inc` au point de rupture. Renvoie l'ancienne valeur du point de rupture.

3 Identification d'un segment

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget( key_t  cle,      /* cle de segment    */
            size_t taille,   /* taille du segment */
            int    option);  /* optionde creation */
```

Renvoie l'identifiant système d'un segment. La taille demandée doit être compatible avec

- les limites imposées par le système en cas de création
- celles du segment s'il existe déjà

4 Opérations spécifiques aux segments

L'opération qui consiste au processus de déplacer le *breakpoint* dans son espace d'adressage pour attribuer à un segment de mémoire partagée une adresse virtuelle est appelée le **attachement**. L'opération inverse, c'est à dire de remettre le *breakpoint* à sa position d'origine quand le processus n'utilise plus le segment s'appelle le **détachement**. La primitive `shmat` réalise l'attachement et la primitive `shmdt` le détachement.

4.1 L'attachement

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat( int         shmid,    /* identification segment */
             const void *adr,      /* adresse demandee */
             int         options);/* option d'attachement */
```

Réalise l'attachement d'un segment d'identification `shmid` à l'espace d'adressage du processus. Renvoie

- l'adresse où l'attachement a été effectivement réalisé si OK
- -1 en cas d'échec

Le principal problème est de choisir une adresse qui n'entre pas en conflit avec les adresses déjà utilisées, qui ne soit pas "à cheval" entre deux pages, qui n'empêche pas l'augmentation de la zone de données et de la pile, ...

4.1.1 Adresse choisie par le système

→`shmat` avec
— `options` = 0
— `adr` = `NULL`
C'est *a priori* la meilleure solution pour avoir des programmes portables.

4.1.2 Adresse choisie par l'utilisateur mais arrondie par le système

→`shmat` avec
— `options` = `SHM_RD`
— `adr` affectée

L'utilisateur peut choisir une adresse virtuelle (avec le paramètre `adr`) en spécifiant qu'elle peut être arrondie si celle ci ne correspond pas à une adresse de segment valide (une adresse valide est un multiple de la constante `SHMLBA`). Si l'attachement est possible, alors il est réalisé à l'adresse `adr - adr % SHMLBA`.

4.1.3 Adresse imposée par l'utilisateur

→`shmat` avec
— `options` ≠ `SHM_RND`
— `adr` affectée
Le système essaie de rattacher le segment à l'adresse `adr` exactement

4.1.4 Autre option

Si le paramètre `options` a pour valeur `SHM_RDONLY` alors le segment est attaché en lecture seulement (c'est à dire que toute écriture déclenchera le signal `SIGSEGV`)

4.2 Le détachement

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(char *adr); /* adresse d'attachement */
```

Effectue le détachement du segment mémoire attaché à `adr`. Cette adresse devient donc illégale pour ce processus.

5 Opérations de gestion des segments

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl( int shmid,                  /* identifiant du segment */
            int op,                     /* operation */
            struct shmid_ds *param );   /* parametres */
```

Effectue l'opération `op` sur le segment `shmid` avec éventuellement les paramètres situés à l'adresse `param`. Renvoie

- 0 en cas de succès
- -1 sinon

Les opérations possibles sont les 3 opérations standards sur les IPC : `IPC_STAT`, `IPC_SET` et `IPC_RMID`. En ce qui concerne `IPC_RMID`, elle a pour effet d'empêcher un nouvel attachement du segment par un processus quelconque. La suppression ne sera effective que lors du dernier détachement.

6 Exemple

Dans cet exemple, le premier processus écrit une série de nombres aléatoires dans un segment de mémoire partagée dont la clé utilisateur est passée en paramètre. Le second processus relit ce segment. Il est à noter qu'aucun des deux processus ne détruit le segment (par la primitive `shmctl` avec l'opération `IPC_RMID`). Pour cela, il faut que l'utilisateur consulte la liste des objets IPC (avec la commande shell `ipcs`) et supprime en ligne le segment (par la commande shell `ipcrm`).

Processus écrivain

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/shm.h>
#define NB 10

int
main( int nb_arg , char * tab_arg[] )
{
     int shmid ;  /* identification du segment */
     int shm_cle ; /* cle utilisateur du segment */
     int * adr ; /* adresse d'attachement */
     int i ;
```

```
        /*-------*/
        if( nb_arg != 2 )
        {
             fprintf( stderr, "usage : %s <cle de segment de memeoire partagee>\n",
                      tab_arg[0] );
             exit(-1);
        }
        if( sscanf( tab_arg[1] , "%d" , &shm_cle) != 1 )
        {
             fprintf( stderr, "%s : erreur , mauvaise cle de segment (%s)\n" ,
                      tab_arg[0]  , tab_arg[1] );
             exit(-2);
        }
        /* Creation du segment ou recuperation de son identifiant */
        if(( shmid = shmget( shm_cle , NB * sizeof(int) , IPC_CREAT | 0666 )) == -1 )
        {
             perror("Pb sur shmget");
             exit(-3) ;
        }
        /* Attachement a une adresse choisie par le systeme */
        if(( adr = shmat( shmid , 0 , 0 )) == (int *)-1 )
        {
             perror("pb shmat") ;
             exit(-4);
        }
        /* Ecriture dans le segment partage */
        srandom(getpid());
        printf("[Ecrivain] Ecriture en cours (E)\n") ;
        for( i = 0 ; i < NB ; i++ )
        {
             adr[i] = random() % 100 ;
             printf("E " ); fflush(stdout) ;
             sleep(random()%5); /* Pour provoquer pb synchro entre lecteur et ecrivain
        }
        printf("\n[Ecrivain] nombre ecrits : " ) ;
        for( i = 0 ; i < NB ; i++ )
          {
            printf("%d " , adr[i] );
          }
        printf("\n") ;
        exit(0);
}
```

Processus lecteur

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

```c
#include <sys/types.h>
#include <sys/shm.h>
#define NB 10

int
main( int nb_arg , char * tab_arg[] )
{
      int shmid ;  /* identification du segment */
      int shm_cle ; /* cle utilisateur du segment */
      int * adr ; /* adresse d'attachement */
      int i ;
      int buf[NB] ;

      /*-------*/
      if( nb_arg != 2 )
      {
            fprintf( stderr , "usage : %s <cle de segment de memeoire partagee>\n",
                     tab_arg[0] );
            exit(-1);
      }
      if( sscanf( tab_arg[1] , "%d" , &shm_cle) != 1 )
      {
            fprintf( stderr , "%s : erreur , mauvaise cle de segment (%s)\n" ,
                     tab_arg[0]  , tab_arg[1] );
            exit(-2);
      }
      /* Creation du segment ou recuperation de son identifiant */
      if(( shmid = shmget( shm_cle , NB * sizeof(int) , IPC_CREAT | 0666 )) == -1 )
      {
            perror("Pb sur shmget");
            exit(-3) ;
      }
      /* Attachement a une adresse choisie par le systeme */
      if(( adr = shmat( shmid , 0 , SHM_RDONLY )) == (int *) -1 )
      {
            perror("pb shmat") ;
            exit(-4);
      }
      /* Lecture dans le segment partage */
      printf("[Lecteur] Lecture en cours (L)\n") ;
      for( i = 0 ; i<NB ; i++ )
      {
        buf[i] = adr[i] ;
        printf("L " ); fflush(stdout) ;
        sleep(random()%5);
      }

      printf("\n[Lecteur] nombre lus : " ) ;
      for( i = 0 ; i<NB ; i++ )
```

```
    {
      printf("%d " , buf[i] ); fflush(stdout) ;
    }
    printf("\n") ;

    exit(0);
}
```

7 Projection des fichiers en mémoire

La version *system V.4* de Unix permet de projeter des fichiers en mémoire. Au lieu de manipuler un fichier à travers les appels systèmes spécifiques (`open, read, write,lseek,close`...), on peut directement y accéder dans l'espace d'adressage du processus, comme le sont les segments de mémoire partagés. Le fichier est vu alors comme un tableau `T` de caractères et le i^{ieme} caractère de ce fichier peut être obtenu par `T[i-1]`. Un fichier projeté en mémoire par plusieurs processus définit un espace partagé par ces processus.

```
#include <sys/mman.h>
void *mmap( void *adr,       /* adresse de projection */
            size_t lg,       /* nb de caracteres */
            int    protection,  /* type d'acces souhaite */
            int    option, /* option adresse */
            int    fd,       /* descripteur fichier */
            off_t  offset); /* debut fichier */
```

Valeurs des paramètres `protection` et `option`

`protection`	Interprétation	`option`	Interprétation
`PROT_READ`	Autorisation de lecture	`MAP_SHARED`	On modifie le fichier lui-même
`PROT_WRITE`	Autorisation d'écriture	`MAP_PRIVATE`	Modif d'une copie privée
`PROT_EXEC`	Autorisation d'exécution	`MAP_FIXED`	à `adr` exactement
`PROT_NONE`	Aucun accès		sinon elle est arrondie