T. Boubekeur, P. Fortin, A. Guermouche, R. Namyst
LaBRI, Bureau 323
le 16/12/2005

Licence d'informatique — Systèmes d'exploitation
Corrigé du Devoir Surveillé du 2/12/2005

1 Gestion mémoire (question d'échauffement)

Dans un système à mémoire *segmentée*, les processus sont divisés en un (petit) ensemble de segments de mémoire contiguë, qui sont placés en mémoire physique de manière indépendante. Par rapport à un modèle où la mémoire des processus est monolithique, cette organisation permet d'atténuer les problèmes de fragmentation mémoire et autorise le partage de certaines zones de mémoire entre processus. Sur ces systèmes, les compilateurs/éditeurs de liens génèrent des adresses relatives au début du segment en fonction de la nature de la données concernée : instruction dans le segment de code, variable globale, emplacement dans la pile, etc. Une adresse (virtuelle) est donc constituée d'un numéro de segment et d'un déplacement au sein de ce segment.

Pour pouvoir effectuer les conversions d'adresses et vérifier la validité des accès, le processeur doit disposer d'un ensemble (registre de base, registre limite, bits de protection) pour chaque segment.

2 Nachos

Question 1 Dans une situation de pénurie de piles, **il ne faut surtout pas** bloquer le *thread appelant* `ThreadCreate` jusqu'à ce qu'une pile se libère, car une telle stratégie pourrait parfois mener à des situations d'interblocage total entre les processus. Par exemple, si on prend l'exemple d'un processus père créant N et que ces $N+1$ processus exécutent tous ensuite le même code contenant une barrière, on voit bien que si le père est bloqué parce qu'il ne peut pas créer le N^{eme} fils, alors les $N-1$ premiers fils seront également bloqués au niveau de la barrière et plus personne ne pourra progresser...

Question 2 La taille de la pile d'un *thread* dépend à la fois des appels de fonction réalisés par ce *thread*, et de l'ensemble des variables locales utilisées (qui sont stockées dans sa pile). Si nous considérons le cas d'un appel de fonction récursive, la profondeur de la récursion dépend de la valeur des paramètres de la fonction : la taille de la pile ne peut alors pas être estimée de façon statique (par exemple lors d'une inspection de code à la compilation) mais seulement de façon dynamique (pendant l'exécution). Cette taille est donc très difficile à borner.

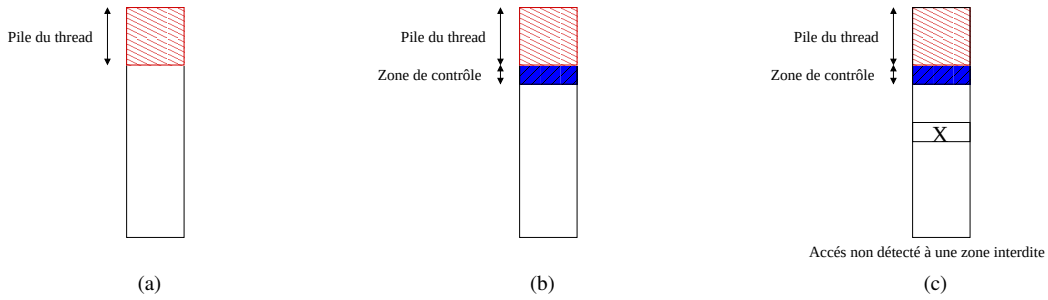

FIG. 1 – Illustration du mécanisme de protection de la pile

Un mécanisme simple pour contrôler les débordements de pile serait d'ajouter à la fin de la pile une *zone de contrôle* initialisée avec une valeur spéciale (*magic number*, par exemple plusieurs 0) à la fin de l'espace de la pile réservé au *thread* (voir figure 1(b)). A chaque changement de contexte, le système peut alors tester le contenu de la zone de contrôle correspondant au *thread* qui perd la main. Si le contenu de la zone a changé, le système arrête le *thread* car il vient d'écrire dans sa zone de contrôle. Notons que cette stratégie ne permet qu'une détection *tardive* car celle-ci ne peut se faire qu'au prochain changement de contexte. Pour détecter un débordement de manière immédiate, il faudrait interdire tout accès mémoire à la zone de contrôle (en modifiant les droits dans la table des pages) pour récupérer le signal `SIGSEGV` par exemple...

Ce mécanisme simple permet de détecter certains débordements de pile mais il est limité. En effet, considérons par exemple la situation illustrée dans la figure 1(c), notons S la taille de la zone de contrôle et considérons la fonction suivante appelée alors que le pointeur de pile est situé juste avant la zone de contrôle :

```
voif f(int param){
  char buffer[S]; /* non utilisé */
  int i=0;
  ...
}
```

Dans ce scénario, le *thread* va modifier une zone mémoire qui est au delà de sa zone de contrôle. Comme le contenu de la zone de contrôle n'a pas changé, le système ne détecte pas alors le débordement de pile.

En pratique, l'utilisation d'une zone de contrôle de grande taille suffit toutefois à détecter la majorité des débordements de pile...

3 Synchronisation

Question 1 Il suffisait de reprendre *littéralement* la solution au problème des producteurs/consommateurs vue en cours :

```
typedef struct {
   ???
   sem_t prod(MAX_PIPE), cons(0), mutex_w(1), mutex_r(1);
} pipe_t;

void write_char(pipe_t *p, char c)
{
   P(p->prod);

   P(p->mutex_w);
   __write_char(p, c);
   V(p->mutex_w);

   V(p->cons);
}

void read_char(pipe_t *p, char *c)
{
   P(p->cons);

   P(p->mutex_r);
   __read_char(p, c);
   V(p->mutex_r);

   V(p->prod);
}
```

Question 2

```
void write(pipe_t *p, char *buf, unsigned len)
{
   unsigned i;

   P(p->mutex_w);
   for(i=0; i<len; i++)
      __write_char(p, buf[i]);
   V(p->mutex_w);
}
```

Question 3

```
void __write_atomic(pipe_t *p, char *buf, unsigned len)
{
   unsigned i;

   P(p->mutex_w);
   for(i=0; i<len; i++)
      __write_char(p, buf[i]);
   V(p->mutex_w);
}

void write(pipe_t *p, char *buf, unsigned len)
{
   unsigned n;

   do {
      n = min(len, MAX_ATOMIC);
      __write_atomic(p, buf, n);
      len -= n; buf += n;
   } while(len > 0);
}
```

Question 4 Voici une solution ne nécessitant pas de modifier le code de `read_char` (il était bien sûr possible de modifier `read_char`, auquel cas on arrivait d'ailleurs à une solution plus simple).

La solution fonctionne, mais son défaut est d'occasionner un peu d'attente active dans certaines situations... La solution sans attente active est laissée en exercice !

```
typedef struct {
   ???
   sem_t prod(MAX_PIPE), cons(0), mutex_w(1), mutex_r(1), sas(1);
   boolean prio=false;
} pipe_t;

void write_char(pipe_t *p, char c)
{
   P(p->sas);
deb:
   P(p->prod);
   if(p->prio) {
      V(p->prod);
      goto deb;
   }
   V(p->sas);

   P(p->mutex_w); __write_char(p, c); V(p->mutex_w);

   V(p->cons);
}

void write_char_urgent(pipe_t *p, char c)
{
   p->prio = TRUE;
   P(p->prod);
   p->prio = FALSE;

   P(p->mutex_w); __write_char(p, c); V(p->mutex_w);

   V(p->cons);
}
```