TP5: Protection mémoire

Matthieu Lemerre, Guillaume Girol, Grégoire Menguy

décembre 2020

Ce TP est la prolongation des TPs 3 et 4 que vous avez fait précédemment. Le but de ce TP est de protéger vos threads (c'est à dire leurs piles) les uns les autres, en utilisant un mécanisme fournit par Linux s'appellant `mprotect`. Ce mécanisme permet au processus de modifier les droits d'accès à certaines de ses pages.

Plus précisément, `mprotect(addr,len,prot)` permet de changer la plage d'adresse entre `addr` et `addr+len-1` pour que les droits correspondents à `prot`. La page de manuel UNIX `man mprotect` vous apprend en détail comment se servir de cette fonction.

Pour utiliser `mprotect`, n'oubliez pas d'inclure le header suivant.

```
#include <sys/mman.h>
```

1 Implantation d'un confinement mémoire

Question 1. *Sur vos machines, la taille d'une page est de 4096 octets. Vérifiez (en début de fonction main) que vos piles commencent à une adresse alignée sur 4096 octet (c'est à dire qui soit un multiple de 4096).*

Question 2. *Modifiez la déclaration de vos piles pour que celles-ci soient alignées en rajoutant* `__attribute__((aligned(4096)))` *en fin de déclaration. Vérifiez que les piles sont maintenant correctement alignées.*

Question 3. *Appellez la fonction* `mprotect` *pour supprimer les droits d'accès aux piles de tous les threads (sauf le thread de l'ordonnanceur, qui reste accessible). Le programme doit maintenant planter lorsque l'un des threads est exécuté.*

Question 4. *Modifiez le code de l'ordonnanceur pour qu'à chaque fois qu'un thread s'exécute, il ait accès à sa pile, mais pas à la pile des autres threads. (Le programme doit maintenant s'exécuter comme à la fin du TP4, sans planter).*

Question 5. *Vérifiez que lorsque vous exécutez un thread, vous ne pouvez pas modifier les donnée d'un autre thread. Pour cela, nous allons injecter une erreur dans le code du producteur: si le caractère que vous tapez est un 'e', vous devez essayer d'écrire dans la pile d'un des consommateurs. Cette opération doit conduire au plantage du programme.*

2 Récupération d'erreur

Le but de cette partie est d'éviter que tout le programme plante quand un des threads fait une erreur mémoire. À la place, nous allons le réinitialiser et le re-démarrer.

Les interruptions permettent au système d'exploitation de réagir (en exécutant un `interrupt_handler`) lorsque certains évènements surviennent, dont une erreur d'accès mémoire (défaut de page).

UNIX (et Linux) propose un mécanisme similaire aux interruptions au niveau du processus, qui s'appelle le signal. Ainsi, lorsqu'un programme commet une erreur mémoire, le signal `SIGSEGV` est envoyé au processus. Il est possible de faire exécuter un traitement spécifique lorsque cela arrive.

Pour cela, vous devez enregistrer le signal handler avec la fonction `signal`, qui prend deux paramètres:

1. Le numéro du signal
2. Une fonction qui prend un argument `int` et qui renvoie `void`, qui sera exécutée lorsque le signal arrive.

Pour utiliser cette fonction, incluez le header:

```
#include <signal.h>
```

Question 6. *Installez un signal handler affichant une information sur le thread qui vient de commettre une erreur mémoire.*

Le but de cette question est d'implanter le redémarrage du thread qui a fait une mauvaise écriture mémoire. Comme l'exécution de ce thread ne peut plus continuer, il faut rendre la main au scheduler dans le signal handler. Et il faut auparavant ré-initialiser le thread qui a planté (avec la fonction `init_thread`).

Question 7. *Implantez la récupération d'erreur par redémarrage lorsqu'un thread plante.*

Note: sur votre système, la récupération d'erreur ne marchera qu'une seule fois (si un thread plante plusieurs fois, les erreurs suivantes ne sont pas récupérées). Cela provient du fait que l'exécution du signal handler est bloquée quand ce handler est exécuté. Pour la débloquer, exécutez le code suivant au début du signal handler.

```
sigset_t set;
sigfillset(&set);
sigprocmask(SIG_UNBLOCK,&set,0);
```