Systèmes d'Exploitation

Devoir 3 : Pagination dans NACHOS

Cette fiche ainsi que les fichiers à télécharger sont disponibles sur le site
https://gforgeron.gitlab.io/se/

L'objectif de ce devoir est de constituer un pas de plus vers la réalisation en Nachos d'un modèle de *processus* à la manière d'Unix, chaque processus pouvant contenir un nombre arbitraire de threads. Plus précisément, il s'agit de mettre en place une gestion mémoire paginée simple au sein du système.

Ce sujet est essentiellement indépendant des deux sujets précédents : vous n'avez essentiellement besoin que d'un `PutChar` et un `CopyStringFromMachine` fonctionnels. Si ce n'est pas le cas, demandez à votre chargé de TD de vous aider à le faire fonctionner. Pour la partie processus+thread, ce sujet dépend bien sûr du sujet précédent, mais il y a déjà beaucoup à faire, ainsi que des bonus, pour éviter d'avoir à dépendre du bon fonctionnement du sujet précédent.

Le devoir est à faire en binôme et à rendre avant

Lundi 9 Décembre 2024, midi.

Comme pour le TP précédent, il vous est demandé de placer votre rapport au sein de votre dépôt git (`projet-xx/rendus/rapport3.pdf`) et de déposer sur Moodle la clé de hash de votre commit.

Avant de commencer à coder, lisez bien chaque partie **en entier** : les sujets de TP contiennent à la fois des passages descriptifs pour expliquer vers où l'on va (et donc il ne s'agit que de lire et comprendre, pas de coder), et des **Actions** qui indiquent précisément comment procéder pour implémenter pas à pas (et là c'est vraiment à vous de jouer).

Partie I. Adressage virtuel par une table de pages

L'ensemble de la machine MIPS travaille en adressage virtuel, selon deux mécanismes au choix de l'utilisateur : la *table des pages* ou le TLB. On ne s'intéressera qu'au mécanisme de la table des pages. Regarder comment elle est initialisée dans `userprog/addrspace.h` et `userprog/addrspace.cc`. Une adresse virtuelle est composée d'un numéro de page et d'un décalage dans la page. À chaque numéro de page virtuel, la table associe un numéro de page physique. Pour l'instant, on met simplement le numéro de page physique `i` pour la page virtuelle numéro `i`, l'adressage virtuel est donc actuellement identique à l'adressage physique. On peut le voir en ouvrant le fichier memory.svg avec firefox : la vue virtuelle (à gauche) affiche la même chose que la vue physique (à droite) car les flèches (la table des pages) sont horizontales. L'adresse (physique) de la page et le décalage dans la page déterminent l'adresse physiquement accédée. Le mécanisme de tranduction est implémenté par la fonction

```
Translate(int virtAddr, int* physAddr, int size, bool writing)
```

dans le fichier `machine/translate.cc`. Tous les accès à la mémoire dans l'interprète se font au travers de cette fonction. Regarder le fonctionnement de `WriteMem` et `ReadMem`.

Action I.1. *Commencer par écrire juste un petit programme de test* `test/userpages0` *qui écrit quelques caractères à l'écran. Il servira de programme de test simple pour la suite.*

Nous allons maintenant charger le programme en mémoire en décalant tout d'une page, mais avant il faut corriger le comportement du chargement du programme.

Action I.2. *Examiner soigneusement l'utilisation de* `executable->ReadAt` *à la fin de la fonction* `AddrSpace::AddrSpace` *de* `userprog/addrspace.cc` *(pas besoin d'aller voir le code de* `ReadAt`*, il s'agit simplement de la combinaison d'un* `lseek` *et d'un* `read`*). Curieusement (?), on lui fait écrire* directement *en mémoire physique MIPS. À quoi voit-on cela ?*

Action I.3. *Définir une nouvelle fonction locale*

```
static void ReadAtVirtual(OpenFile *executable, int virtualaddr,
int numBytes, int position, TranslationEntry *pageTable,
unsigned numPages)
```

qui aura le même effet que `ReadAt` *(c'est-à-dire lire* `numBytes` *octets depuis la position* `position` *dans le fichier* `executable`*), mais en écrivant dans l'espace d'adressage virtuel défini par la table des pages* `pageTable` *de taille* `numPages`*. Vous pouvez utiliser un tampon temporaire dans l'espace noyau, que vous remplirez en appelant* `ReadAt` *une seule fois, puis que vous recopierez en mémoire MIPS octet par octet avec* `WriteMem` *par exemple (même si c'est peu efficace ; n'essayez pas d'optimiser, c'est très difficile)... Pensez bien à changer temporairement de table de page dans la machine, pour que* `WriteMem` *utilise bien la table de pages construite dans le constructeur* `AddrSpace::AddrSpace`*, et restaurez-la correctement (inspirez-vous de* `space->RestoreState`*).*

Utilisez `ReadAtVirtual` *en lieu et place de* `ReadAt` *là où c'est nécessaire, et vérifiez que l'exécution de programmes fonctionne toujours, notamment utilisant* `PutChar` *et* `PutString`*. Sinon, vérifiez dans* `memory.svg` *comment le programme est chargé.*

Action I.4. *Modifiez la création de la table des pages pour que la page virtuelle* i *soit une projection de la page physique* $i + 1$ *(pour stresser votre implémentation sur un cas simple pour commencer et pouvoir facilement débugguer)*

Relancez votre programme. Tout doit marcher, et les threads utilisateurs doivent s'exécuter normalement !

Observez les traductions d'adresse avec l'option de trace `-d a`*.*

Vous pouvez également regarder le fichier `memory.svg` *produit.*

Plus généralement, il est utile d'encapsuler l'allocation des pages physiques dans une classe spéciale `PageProvider` globale à tout nachos. Notez que puisque l'on réutilisera les pages physiques cette classe devra remettre à zéro le contenu des pages allouées.

Action I.5. *Créer une classe* `PageProvider` *dans le fichier* `userprog/pageprovider.cc` *qui s'appuie sur la classe* `BitMap` *pour gérer l'allocation des pages physiques (à chaque bit du bitmap correspond une page physique). Elle permet : 1) méthode* `GetEmptyPage` *: de récupérer le numéro d'une page physique libre* et d'initialiser le contenu de la page physique correspondante à `0` grâce à la fonction `memset` *; 2) méthode* `ReleasePage` *: de libérer une page obtenue précédemment par* `GetEmptyPage` *; 3) méthode* `NumAvailPage` *:*

de demander combien de pages restent disponibles. Notez que la politique d'allocation des pages est complètement locale à cette classe. Pourquoi faut-il un seul objet de cette classe `PageProvider`*? On pourra donc le créer en même temps que la machine dans* `Initialize`*.*

Action I.6. *Corriger le constructeur et le destructeur d'*`AddrSpace` *pour utiliser ces primitives, et faites tourner votre programme avec diverses stratégies d'allocation. Par exemple, juste pour tester et stresser l'implémentation, allouer les pages par un tirage aléatoire!*

Vous pouvez appeler `machine->DumpMem("addrspace.svg");` *à la fin du constructeur* `AddrSpace` *pour observer dans* `addrspace.svg` *l'allocation des pages et leur remplissage.*

Quel type d'erreur peut survenir? Pensez à le traiter! (Mettez au ***minimum*** *un ASSERT) Si vous avez des soucis, pensez à utiliser l'option* `-d a` *pour observer les traductions d'adresses.*

Vérifiez bien que tous vos programmes fonctionnent toujours.

Partie II. Exécuter plusieurs programmes en même temps...

Puisque désormais seule une partie de la mémoire physique est utilisée pour projeter les pages virtuelles (`size` n'est pas forcément égal à `MemorySize`), pourquoi ne pas conserver dans la mémoire *plusieurs* programmes en même temps?

Action II.1. *Définir un appel système* `int ForkExec(const char *s)` *qui prend un nom de fichier exécutable et lance un nouveau processus qui exécute ce programme dedans.*

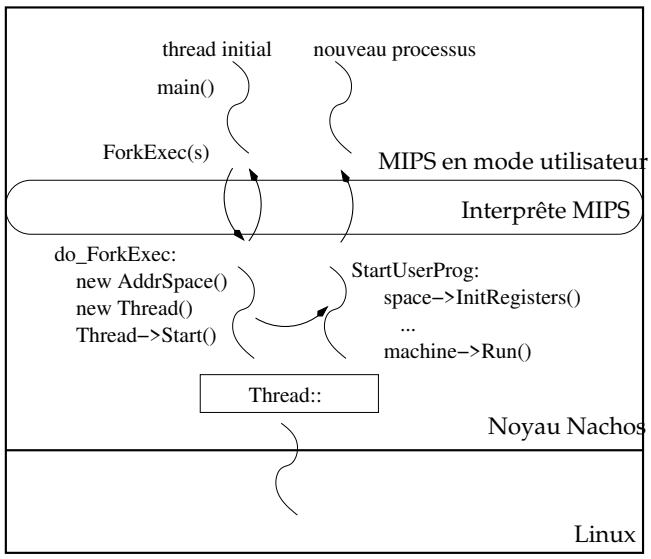
Il s'agit donc en fait de faire l'équivalent de `StartProcess`*, mais en deux morceaux, de manière analogue à la création de thread du TP2. Comme montré sur le dessin plus bas, il s'agit de commencer dans le père par créer un objet* `AddrSpace` *à partir de ce fichier exécutable, créer un thread noyau, définir son* `space` *à l'*`AddrSpace` *nouvellement créé, et le faire lancer une nouvelle fonction* `StartUserProc`*. Cette dernière fait le deuxième morceau : initialiser les registres et lancer l'exécution du nouveau processus utilisateur, en parallèle avec le processus père.*

Le programme suivant doit donc fonctionner. Il faudra éventuellement augmenter `NumPhysPages`*. Dans un premier temps, mettez des* `while(1);` *à la fin de* ***toutes*** *vos fonctions* `main` *et activez la préemption avec* `-rs`*, pour laisser le temps aux différents processus de faire leurs affichage. Vous règlerez les problèmes de terminaison plus tard.*

Si vous avez des bugs, vérifiez bien que le nouveau space est bien attribué au bon processus, que la bonne table des pages est bien chargée au bon moment.

```
#include "syscall.h"
main()
{
  ForkExec("../test/putchar");
  ForkExec("../test/putchar");
  while(1);
}
```

Vous pouvez observer dans `addrspace.svg` *les différents espaces d'adressage.*



Action II.2. *Raffiner votre implémentation pour que lorsque le dernier processus s'arrête, un appel à* `interrupt->Powerdown()` *soit effectué automatiquement, i.e. l'appel système* `Exit()` *ne doit plus désormais systématiquement appeler* `interrupt->Powerdown()` *: s'il reste d'autres processus, il ne faut que terminer le processus courant pour laisser les autres tourner. Notez bien que l'on ne traite pas encore les threads ici, ils seront l'objet de l'action ci-dessous. Pensez par contre à libérer* immédiatement *toutes les ressources qu'il utilise (sa structure* `space`*, etc.).*

Action II.3. *Notez maintenant que le programme courant et le programme lancé peuvent eux-mêmes contenir des threads (lancé par une fonction du niveau MIPS)! Le programme ci-dessous doit donc fonctionner.*

Vous aurez éventuellement à déplacer des variables globales dans la classe `AddrSpace` *pour en avoir un exemplaire par processus, et corriger notamment le comptage des threads.*

Pour simplifier dans un premier temps, vous pouvez supposer que les programmes MIPS ne mélangent pas threads et Exit : soit ils n'utilisent pas de threads et donc terminent avec Exit, soit ils utilisent des threads et alors tous les threads appellent gentiment `ThreadExit` *(y compris le thread principal),* `Exit` *n'étant alors jamais appelé. Le mélange des deux sera traité dans le bonus II.5.*

```
#include "syscall.h"

main()
{
  ForkExec("../test/userpages0");
  ForkExec("../test/userpages1");
}
```

avec pour `userpages0` *et* `userpages1` *des programmes du genre*

```
#include "syscall.h"
#define THIS "aaa"
```

```c
#define THAT "bbb"

const int N = 10; // Choose it large enough!

void puts(const char *s)
{
  const char *p; for (p = s; *p != '\0'; p++) PutChar(*p);
}

void f(void *arg)
{
  const char *s = arg;
  int i;
  PutChar('x');
  for (i = 0; i < N; i++)
    puts(s);
  ThreadExit();
}

int main()
{
  ThreadCreate(f, THIS);
  f(THAT);
  ThreadExit();
}
```

Action II.4. *Montrez que vous pouvez lancer un grand nombre de processus (disons une douzaine), chacun avec un grand nombre de threads (une douzaine aussi, si vous l'avez implémenté au TP2).*

Action II.5. *(Bonus) Mélangeons maintenant threads et Exit ! Lorsqu'un thread d'un processus appelle Exit, on doit terminer tous les threads de ce processus (et terminer le processus). Il faut donc conserver la liste des threads d'un processus quelque part...*

Action II.6. *(Bonus) Si l'un des threads ainsi détruit de manière brutale était en train d'effectuer un* `PutString`*, a priori il détenait encore le verrou que vous avez ajouté pour éviter que les* `PutString` *se mélangent entre threads. Comment corriger cela ?*

Action II.7. *(Bonus) Montrer en surveillant la consommation des ressources de votre processus Unix propulseur que les processus MIPS/Nachos libèrent effectivement bien leurs ressources une fois terminés. Pour cela, vous pouvez utiliser valgrind :*

```
valgrind --leak-check=full
```

qui vous indiquera les zones non libérées. Il se peut qu'il râle pour la pile du dernier thread noyau en cours d'exécution, c'est normal, pourquoi ?

Partie III. Bonus : shell

Action III.1. *Implémentez un tout petit shell en vous inspirant du programme* `test/shell.c`*.*