Martial DUVERNEIX
Teiki PEPIN

Rapport
Systèmes d'exploitation
Projet 3 : Pagination

I – Bilan :

Dans le cadre de la mise en place d'une gestion mémoire paginée, nous avons commencé par l'étude du mécanisme de la table des pages sous Nachos. La table des pages est allouée et initialisée dans le fichier userprog/addrspace.*h* via les fonctions *AddrSpace(OpenFile * executable)*.

Nous observons dans l'exécution de la fonction *ReadAt(...)* que le programme écrit directement en mémoire physique MIPS dû à l'utilisation de *machine→mainMemory*. Nous définissons une nouvelle fonction *ReadAtVirtual(…)* qui nous permet d'écrire les valeurs lues par *ReadAt(…)* dans la mémoire virtuelle via la fonction *WriteMem(…)*. Elle fait cela en maintenant la table de pages courante.

Nous modifions ensuite la création de la table des pages telle que chaque page virtuelle soit équivalente à la page physique suivante.

Pour continuer, nous encapsulons l'allocation de pages physiques à une nouvelle classe *PageProvider*. Cette dernière comporte une *BitMap* qui correspond à l'allocation courante des pages et implémente des fonctions basiques sur cette *BitMap* : *PageProvider(int), ~PageProvider, int GetEmptyPage(), void ReleasePage(int), int NumAvailPage()*. Il ne faut qu'un seul objet de la classe *PageProvider,* car les pages sont partagées par tous les processus. Nous l'initialisons donc dans *system.cc.*

Dans le constructeur d'*AddrSpace*, nous lui attribuons une page physique en fonction des pages disponibles dans le *PageProvider*. De même, nous libérons toutes les pages du *PageProvider* dans le destructeur de *AddrSpace*. Un *assert* a été ajouté pour le cas où le *PageProvider* ne disposerait pas d'aucune page disponible.

Nous créons ensuite un appel système *ForkExec* en modifiant les fichiers *syscall.h, start.S, exception.cc*. Celui-ci est défini dans *userprocess.cc* où il crée un nouvel espace d'adressage et l'attribut un nouveau thread.

Nous avons implémenté dans *system.cc* un compteur de processus et les fonctions correspondantes : *IncCounterProcess(), DecCounterProcess()* et *IsProcessRunning()*. Nous nous servons de ces fonctionnalités lors de l'appel système *Exit* afin de mettre fin au thread courant si d'autres processus sont toujours actifs et sinon d'appeler *Halt()*. L'espace d'adressage du thread appelant est libéré dans les deux cas.

De manière similaire, nous implémentons ces fonctionnalités dans *userthread.cc* afin que la gestion des processus soit effectuée lors de l'appel de *ThreadExit*. Nous pouvons désormais lancer un grand nombre de processus qui lancent eux-mêmes un grand nombre de threads de telle façon que chaque thread se termine avec soit *Exit* ou *ThreadExit*. Pour cela, il est nécessaire d'augmenter le nombre de pages physiques *NumPhysPages* qui est dans le fichier *machine.h* (nous l'avons mis à 256) et de forcer l'ordonnancement préemptif des threads utilisateurs.

II – Points délicats & limitations :

Nous avons rencontré des difficultés dans l'allocation de pages physiques du *PageProvider* par un tirage aléatoire lors de la génération d'un nombre aléatoire.

Dans le cas où le nombre de pages physiques attribuées à Nachos ne serait pas suffisant pour l'exécution des tests désirés, Nachos va se terminer lorsque l'assertion du nombre de pages disponibles dans *PageProvider* échoue.

Dans la situation où l'ordonnancement préemptif des threads utilisateurs n'est pas forcé, la création d'une grande quantité de threads entraîne une boucle infinie due au fait que les threads ne sont jamais libérés, car le thread parent conserve la main dans sa tentative de création de threads.

Il faut noter que, avec notre implémentation, les programmes se terminent soit avec *Exit* soit avec *ThreadExit*, mais ne peuvent mélanger les deux.

III - Tests :

Au début de la première partie, nous avons mis en place un programme de test dans *userpages0.c* qui lance plusieurs *PutChar* successifs tel que demandé dans l'énoncé.

Après l'implémentation de *PageProvider* dans la première partie, tous nos programmes de test précédemment créés fonctionnent toujours.

Notre implémentation initiale de *ForkExec* nous permet de lancer d'autres processus qui ne se terminent jamais via une boucle infinie.

Une fois la gestion de processus et de threads implémentée, nous nous servons de nos programmes utilisateurs pour lancer 12 processus qui lancent chacun 12 threads, soit un total de 144 exécutions, qui affiche des chaînes de caractères suivies d'un caractère '-' (*./userprog/nachos -rs 123456789 -x ./test/forkexec > test.txt*). Nous utilisons *grep* pour confirmer que les 144 exécutions s'effectuent correctement (*grep - -o test.txt | wc -l*).