Rapport TD3 OS

Valentin Jonquière, Mathilde Chollon

8 décembre 2024

Table des matières

1 Bilan

Pour ce troisième rendu, nous avons implémenté tous les bonus. Nous avons une nouvelle fois codé tout le début du projet ensemble, souvent sur un seul ordinateur. Nous nous sommes répartis le travail sur certains bonus. Les parties du code qui ne nous ont pas posé de problèmes particuliers ne sont pas mentionnées dans ce rapport.

2 Points Délicats

2.1 Partie I

2.1.1 Action I.2

Nous voyons que nous écrivons en mémoire Mips car il n'y a pas de translation d'adresses. De plus, lorsque l'on compare la mémoire physique et la mémoire virtuelle grâce à *DumpMem*, nous pouvons voir que les adresses sont les mêmes.

2.1.2 Action I.5

La création de la classe *PageProvider* n'a pas vraiment posé de problème. Nous avons surtout réfléchi à quels paramètres donner à cet objet et que sauvegarder. Nous avons adapté notre implémentation au fur et à mesure que nous trouvions des paramètres à rajouter. Nous avons décidé de sécuriser notre implémentation avec des *Mutex* dès le départ, même si l'arrivée des threads était dans les actions suivantes. Il ne faut qu'un seul *PageProvider* car si chaque espace d'adressage avait son propre *PageProvider*, il pourrait supprimer des pages qui ne lui appartiennent pas et il n'y aurait pas d'entente globale sur qui possède quelle page, et donc quelles pages sont libres pour un nouveau processus.

2.1.3 Action I.6

Nous avons passé beaucoup de temps sur cette action. Au départ, la pagination se faisait correctement, mais nous avons rencontré un problème lorsque nous avons tenté d'implémenter la pagination aléatoire. En effet, lorsque nous lancions notre programme, selon l'option utilisée pour l'ordonnanceur (-rs), nous avions des SegFaults. Il était écrit que nous tentions d'accéder à de la mémoire en dehors des adresses physiques (ce qui n'était pas possible du point de vue de notre fonction *getRandomPage()*). Nous avions en fait tenté de faire un *Translate* dans notre fonction *ReadAtVirtual* ce qui provoquait l'erreur. L'adresse physique obtenue n'était même pas utilisée, ce bout de code ayant été fait avant d'avoir totalement compris la fonction. Nous avons également protégé la demande des pages par deux *ASSERT*, le premier à pour objectif de vérifier que le nombre de pages physiques est suffisant pour initialiser un *AddrSpace*. Un deuxième vérifie que le *PageProvider* nous retourne une page valide (différente de -1) car si deux processus demandent des pages en même temps il est possible qu'il n'y ait pas assez de pages disponibles pour les deux processus même si le

premier assert est passé (car demander le nombre de pages libres ne les réserve pas). Nous avons ensuite essayé plusieurs modes d'allocation des pages que nous avons pu visualiser à l'aide des *DumpMem*. Pour commencer une allocation basique qui fait correspondre les adresses physiques aux adresses virtuelles qui donne la figure 2 lorsque l'on exécute le programme *alloctypescomparison*. Nous avons également testé une allocation basique, mais qui décalait toutes les pages d'une page. Pour finir, nous avons utiliser une méthode d'allocation aléatoire dont le résultat peut être visualisé dans la figure 3 (qui est également l'image de l'exécution de *alloctypescomparison*).

2.2 Partie II

2.2.1 Action II.1

Cette partie ne nous a pas posé de problème en particulier, car cela était très similaire à la création de nouveaux threads et au code de *StartProcess*. Nous avons pu vérifier que nous arrivions à créer de nouveaux processus grâce au *DumpMem*. En effet, cela nous a permis de voir que plusieurs processus étaient présents dans la mémoire virtuelle et d'observer les correspondances dans la mémoire physique. Nous avons testé les limites de notre implémentation en cherchant le maximum de processus pouvant être lancés en même temps, avec différents nombres de pages physiques.

2.2.2 Action II.5

Afin de sauvegarder la liste des threads encore vivants d'un processus, nous avons créé un tableau de threads dans *AddrSpace*. Il est de la taille du maximum de threads pouvant être créé. En effet, comme chaque processus a son propre espace d'adressage, c'est l'endroit où garder cette information. Lorsqu'un thread d'un processus appelle *Exit*, nous faisons maintenant une boucle sur tous les threads vivants de ce processus, et nous les supprimons.

2.2.3 Action II.6

Nous avons décidé d'implémenter une nouvelle fonction noyau pour les *Locks* : *ForceRelease*. Lorsque nous supprimons tous les threads d'un processus comme implémenté à l'action II.3, nous regardons si l'un d'entre eux possède le lock de *PutString* ou de *GetString*, et nous relâchons le lock de manière brutale, sans avoir à être le thread ayant acquis le mutex.

2.3 Partie III

Pour créer un shell basique, il a fallu trouver un moyen d'attendre la fin de l'exécution d'un processus. En effet, l'idée était de simplement créer une boucle infinie qui attendait un programme à exécuter, l'exécute et attend la fin de l'exécution pour recommencer. Pour pouvoir attendre la fin de l'exécution, nous avons donc implémenté les *Conditions* dans *synch.cc* (en nous inspirant

des locks et des sémaphores). Nous avons ensuite implémenté un nouvel appel système *Wait* qui permet à un thread d'attendre d'être le dernier processus actif sur la machine. Pour que cette condition soit commune à tous les processus de la machine, nous l'avons ajoutée au *PageProvider* et avons également ajouté un lock spécifique à cette condition (toujours dans le *PageProvider*). Cet appel système est dangereux, car on peut facilement bloquer la machine en faisant deux appels *Wait* dans des processus différents (aucun des deux ne se réveillera puisqu'ils attendent tous les deux d'être l'unique processus en exécution). Cependant, c'est l'appel dont on a besoin pour réaliser notre shell. Nous avons essayé de nous rapprocher de la fonction *pid_t wait(int *_Nullable wstatus);* du langage C, mais basée sur les processus et non pas les threads.

3 Limitations

3.1 Partie I

Nous avons décidé de garder l'allocation des pages aléatoire avec le *PageProvider*. Ce n'est pas l'implémentation la plus optimisée puisque lorsque la mémoire se remplit, nous mettons plusieurs itérations avant de trouver une page libre. Cependant, nous avons préféré le laisser ainsi, car avec l'allocation linéaire, il y avait plus de possibilités que les accès mémoire des threads, locks, processus,... fonctionnent par "chance". Avec l'allocation aléatoire, il est beaucoup plus simple de trouver des erreurs si elles sont présentes.

3.2 Partie II

3.2.1 Action II.6

La fonction *ForceRelease* que nous avons implémentée casse le principe des locks, qui assure que seul le thread possédant le mutex peut le relâcher. C'est une solution brutale, qui n'a donc pas été implémentée coté utilisateur. Nous partons du principe que cette fonction ne doit être utilisé que lorsqu'il n'y a pas d'autre choix. Nous avions également pensé à une autre solution : modifier le *Program Counter* du thread possédant le mutex pour le placer à l'adresse de *Release*. Cette solution étant beaucoup moins élégante et surtout plus propice à des erreurs lors de la modification du registre, nous l'avons rapidement écartée.

3.2.2 Action II.7

Comme discuté dans le rapport 2, la pile du dernier thread n'est pas libérée, car il ne peut pas se supprimer lui-même. Il y a aussi des fuites mémoires au niveau du dernier *AddrSpace* car on ne peut pas effectuer le *Cleanup* de celui-ci. En revanche les autres espaces d'adressages créés pour les différents threads sont bien supprimés.

3.3 Partie III

Les principales limitations de notre implémentation sont dues à l'appel système *wait* que nous avons créé. Par exemple l'appel *wait* rend impossible le fait de quitter le shell via un programme (car *wait* attend d'être seul et aucun *exit* ne peut mettre fin à l'exécution du processus principal). L'unique moyen de terminer le programme était donc de faire un appel à *halt*, mais l'utilisation de cet appel système empêche une terminaison propre où la mémoire est correctement libérée. Nous avons défini que si la commande donnée par l'utilisateur est simplement un 'e', le shell faisait un appel à *exit* pour terminer de manière 'basique'. L'appel système *wait* ne nous permettant pas d'attendre la fin de l'exécution d'un processus en particulier, nous sommes obligés d'attendre que le shell soit seul à être exécuté sur la machine et nous ne pouvons pas lancer de commande en arrière-plan par exemple. Il est aussi discutable que seul un thread attende de faire partie de l'unique processus, car nous n'avons pas géré le cas ou ce thread se ferait tuer si son processus terminait. D'autres fonctionnalités basiques comme la redirection de l'entrée ou sortie standard vers des fichiers ou encore les pipes ne sont donc pas disponibles. Nous avons également trouvé un bug mineur (uniquement graphique), si l'on tente d'exécuter un programme qui n'existe pas alors la console passera en rouge et ne reviendra pas au blanc, cela n'a cependant aucun impact sur le comportement du programme.

3.4 Fuites Mémoires

Les tests écrits combinés à *Valgrind* nous ont permis d'identifier de nouvelles fuites mémoires. Même si nous avons pu corriger la plupart, car il s'agissait surtout d'objet C++ comme des *Locks* que nous oublions de supprimer à la destruction du *PageProvider*. Il reste toujours des fuites qui (de part l'implémentation choisie) ne pourrons pas être corrigées. Nous allons prendre l'exemple d'une fuite qui a lieux lors de l'exécution du test *forcereleasetest* où *Valgrind* nous annonce une perte définitive de 5 octets. Pour l'expliquer nous avons pris une capture d'écran du code responsable(Figure 1 en annexe). Comme on peut le voir, la ligne annotée 1 alloue bien un buffer dynamiquement (celui-ci devrait être rempli lors de l'appel a *GetString*). Dans l'étape 2, le thread va bien exécuter *GetString* et se mettre à attendre des caractères frappés au clavier. Cependant, pendant son attente le deuxième thread du processus va lui terminer le processus (et donc lancer la procédure de destruction des threads et de *force-release* des locks *PutString* et *GetString*). On voit donc que le thread ne reviendra jamais de l'appel numéroté 2 et, par conséquent, n'exécutera jamais l'instruction 3 qui est la libération du buffer (le *copyStringToMachine* n'est, lui aussi, jamais exécuté dans ce cas). Nous pouvons donc noter que ce type de fuites mémoires est lié au choix d'implémentation et ne peut être corrigé facilement. Il reste cependant très minoritaire, puisque dû à l'appel de *force-release* qui est une procédure particulière.

4 Tests

Nos tests sont décrits en détail dans des commentaires présents dans leurs fichiers. De manière générale le test *multipleprocesses* est celui que nous avons utilisé. Il présente les caractéristiques nécessaires pour vérifier qu'un processus fonctionne (il crée des threads dans chaque processus qu'il crée). Il est cependant nécessaire d'augmenter grandement le nombre de pages physiques disponibles (*NumPhysPages* l.38 dans *machine.h*) ainsi que la taille par défaut de la pile du processus (*UserStacksAreaSize* l.26 dans *addrspace.h*).

4.1 Partie I

4.1.1 Action I.3

Afin de tester notre nouvelle fonction *ReadAtVirtual*, nous n'avons pas implémenté de nouveaux tests. Nous avons utilisé nos anciens programmes qui utilisaient *PutString* et *PutChar* afin de vérifier qu'il n'y avait pas de problème de lecture à de mauvais endroits de la mémoire.

4.1.2 Action I.6

Lorsque nous avons commencé nos tests, nous avions beaucoup d'erreurs de segmentation. En effet, vu que nous faisions un *Translate* qui n'avait pas lieu d'être, on se retrouvait souvent avec des valeurs incohérentes qui faisaient planter notre programme. Après avoir corrigé ce bug, nos tests se sont remis à fonctionner. Nous avons également créé un test *alloctypescomparison* qui créé quelques processus contenant des threads afin de visualiser les effets des différents types d'allocation sur la mémoire physique/virtuelle. Pour changer de méthode d'allocation, il suffit de dé commenter la ligne 142 de *addrspace.cc* et de commenter la ligne suivante (Pour passer à l'allocation classique).

4.2 Partie II

4.2.1 Action II.1

Afin de tester notre première implémentation de *ForkExec*, nous avons utilisé le programme de test noté dans le sujet. Nous n'avions pas observé de bug et sommes donc passés à la partie suivante afin de pouvoir tester nos programmes avec la terminaison.

4.2.2 Action II.2

Nous n'avons pas implémenté de nouveaux tests pour cette partie, nous avons seulement retiré le *while(1)* du test de l'action précédente et appelé d'autres programmes de test pour observer les retours.

4.2.3 Action II.3

Pour tester l'utilisation des threads dans les différents espaces d'adressage, nous avons utilisé le test présent dans le sujet.

4.2.4 Action II.6

Pour tester cette action, nous utilisions le test *forcerelease*, qui appelle un autre programme : *testrelease*. Nous avons décidé de relâcher les verrous de *PutString* et *GetString* en cas de terminaison brutale d'un processus. Ici, nous avons décidé de tester le release de *GetString*. Pour tester notre implémentation, nous avons testé l'utilisation de *ForceRelease* (fonction non accessible coté utilisateur). Nous avons créé un programme qui crée un processus en appelant *ForkExec* et qui créait un thread appelant *GetString*. Le processus crée créait deux threads qui appelaient *GetString* et *PutStrig*. Dans la fonction appelant *PutString*, nous faisons un *Exit* afin de supprimer brutalement le processus lorsque la fonction termine. Ce test nous a permis de modifier notre implémentation afin de ne pas se retrouver dans un deadlock ou de tenter de réveiller un thread tué.

4.3 Partie III

Le shell basique que nous avons codé se trouve dans le fichier *shellV2.c*. Vous pouvez lancer n'importe quel programme contenu dans le dossier test. Il n'était pas possible de faire de réel test avec le shell. Cependant, tous les programmes que nous avons écrits et testés ont eu le même comportement que lorsqu'ils sont lancé directement avec nachos. On peut donc considérer que celui-ci est correct (il gère les processus lançant plusieurs processus et plusieurs threads). Nous avons tout de même un problème avec les *DumpMem* dans notre shell d'après *Valgrind* puisque l'on trouve des 'Conditional jump or move depends on uninitialised value(s)' lors de l'exécution d'un nouveau programme. Nous n'avons cependant pas cherché à le corriger par manque de temps et, car ce message ne mentionne que les *DumpMem* (si on commente les *DumpMem* il n'y a pas de problème de mémoire).

5 Annexes

FIGURE 1 – Code responsable des fuites mémoires dans le test *forcereleasetest*

```
case SC_GetString:
  {
    DEBUG ('s', "[SYSCALL] GetString\n");
    char* buffer = (char*)malloc(sizeof(char)*machine->ReadRegister(5));  (1)
(2) consoledriver->GetString(buffer, machine->ReadRegister(5));
    copyStringToMachine(buffer, machine->ReadRegister(4), machine->ReadRegister(5));
(3) free(buffer);
    break;
  }
```

FIGURE 2 – Exécution du programme de test *alloctypescomparison* avec allocation classique

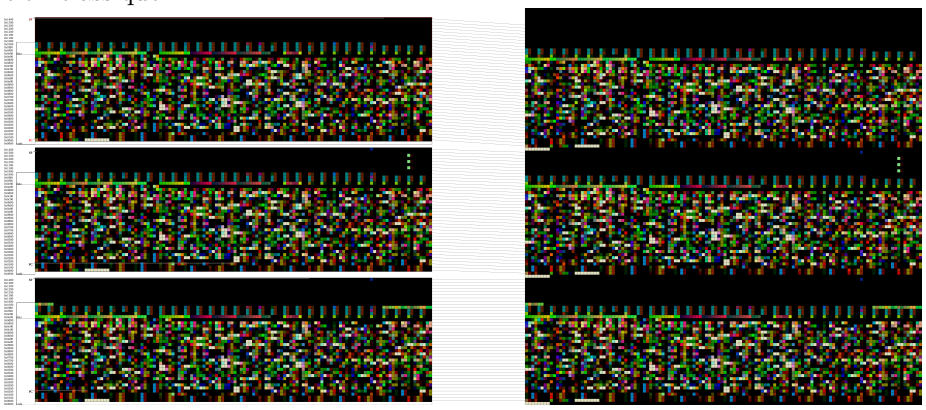

FIGURE 3 – Exécution du programme de test *alloctypescomparison* avec allocation aléatoire