Rapport TD2 OS

Valentin Jonquière, Mathilde Chollon

11 novembre 2024

Table des matières

1 Bilan

Toutes les actions semblent fonctionner dans l'ensemble, mais nous avons rencontré un problème avec l'utilisation de *printf*. En effet, cette fonction ne gère pas correctement les boucles *for*, ce qui rend son utilisation compliquée dans notre projet. Pour contourner cette limitation, nous avons utilisé *PutString* et *PutInt* à la place de *printf* dans nos tests et nos implémentations, ce qui a permis d'éviter des erreurs inattendues lors de l'affichage. Nous avons tenté de le modifier en le sécurisant avec des sémaphores, mais sans succès. Notre code de *vsprintf.c* se trouve dans l'espace utilisateur et ne comporte pas de main où créer le sémaphore comme dans nos programmes de tests. Nous ne pouvions pas non plus faire comme pour *PutString* car nous ne sommes pas en espace noyau. La partie I ne nous a pas particulièrement posé de problèmes, nous ne nous sommes pas attardés dessus dans le rapport.

2 Points Délicats

2.1 Partie II

Nous avons passé du temps sur la création de mutex. En effet, nous avons commencé par les créer à l'aide des sémaphores initialisés à 1 afin de ne pas copier/coller la quasi-totalité de leurs fonctions. Après une discussion avec notre chargé de TD, nous avons décidé d'implémenter les mutex sur la même base de code que les sémaphores, en mettant un booléen *locked* pour représenter l'état du mutex à la place de l'entier *value* ainsi qu'en rajoutant un pointeur vers le thread ayant verrouillé le mutex.

Nous avons ensuite utilisé ces mutex afin de sécuriser *PutChar* et *GetChar*. Pour cela, nous avons utilisé deux mutex, *writeLock* et *readLock*. Il fallait utiliser deux mutex car nous pouvons faire *PutChar(GetChar)* par exemple, ce qui ne fonctionnerait pas avec seulement un mutex. Nous avons également protégé *PutString* et *GetString*. En effet, sans protection, nous pouvions être interrompus lors d'un changement de contexte par un autre thread utilisant également *PutString*, ce qui entremêlerait les deux strings.

Avant de faire l'*Action II.3*, si nous lancions plusieurs threads, ils avaient la même pile. Nous ne pouvions pas allouer dynamiquement les threads, car nous ne savions pas où se trouvait la dernière pile allouée. Nous sommes donc passés à l'*Action II.4*. Nous avons délégué ces opérations à l'*Address Space*. Nous avons créé un *Bitmap* de taille égale au nombre de threads utilisateurs pouvant être créés par rapport à la taille de la pile allouée pour les stack des utilisateurs (*UserStacksAreaSize/256*). Avant chaque création de thread, nous vérifions si nous avons un espace de libre pour la stack et nous créons le thread que s'il y a de la place.

2.2 Partie III

Si un thread n'appelait pas *ThreadExit*, il ne pouvait pas se détruire. Pour éviter cela, nous avons implémenté la terminaison automatique (mis à part pour le thread *main*). Nous avons utilisé une autre méthode que celle donnée dans le TD en plaçant directement l'adresse de la fonction *ThreadExit* dans le registre 6 (pour le passer comme argument) et rendre le code indépendant de l'adresse de chargement de la fonction *ThreadExit*.

2.3 Partie IV

Lorsque nous avons voulu remonter l'utilisation des sémaphores vers l'espace utilisateur, nous avons beaucoup réfléchi à l'implémentation. Nous avons rajouté quatre appels systèmes : *SemaphoreCreate*, *SemaphoreDelete*, *P* et *V*. Les threads d'un même espace d'adressage ont accès aux mêmes sémaphores. Pour cela, nous avons un *Bitmap* afin de savoir quels sémaphores sont créés et un tableau de Sémaphores pour les stocker. L'utilisateur a uniquement accès à l'indice de son sémaphore. Lorsque nous avons commencé à écrire nos fonctions P et V dans *addrspace.cc*, nous avions protégé les fonctions à l'aide d'un mutex. Cela nous a mené à des deadlocks lorsque nous avons testé l'implémentation avec le test producer-consumer. Nous avons donc retiré ce mutex, en effet, les fonctions P et V de *synch.cc* gèrent déjà la concurrence des threads.

2.4 Tests

Lorsque nous avons fait nos tests, nous avons eu un problème avec notre *printf*. En effet, nous nous sommes rendus compte qu'il ne gérait pas les boucles for. Nous n'avons pas réussi à expliquer le comportement de notre fonction et avons donc choisi d'utiliser des *PutString* et *PutInt* à la place.

3 Limitations

3.1 Partie I

Beaucoup de memory leak avec les threads : les fonctions de Cleanup ne suppriment pas la stack. En effet, il y a bien une fonction *StackAllocate* mais pas de *StackDeallocate*. Nous nous retrouvons donc avec 73K octets non supprimés à la fin de l'exécution lorsque nous utilisons des threads.

3.2 Partie II

Nous utilisons la variable *UserStacksAreaSize* du fichier *addrspace.h* afin de créer les piles de nos threads. Nous avons gardé la valeur de départ 1024, ce qui nous donne quatre piles de 256 octets. L'utilisateur ne peut donc avoir que 4 threads en simultané. Si nous avons besoin de plus, il suffit de modifier la

variable. Nous avons gardé cette valeur, car il était plus simple de se rendre compte de possibles erreurs avec 4 threads.

Lorsque nous n'avons pas de place pour créer un nouveau thread, on ne le crée pas et on renvoie *-1*. Il serait peut-être judicieux d'avoir une liste d'attente pour les threads non crées par manque de place. En effet, si nous voulons créer un thread, mais qu'il n'a pas de place, il ne se créera pas et donc n'exécutera jamais son code.

3.3 Partie IV

Concernant les sémaphores utilisateurs, nous avons choisi d'avoir un bitmap de taille 16, donc 16 sémaphores en simultané pour un même espace d'adressage. Cette limite est modifiable dans le fichier *addrspace.cc* à *MAX_SEMAPHORES*, dans le cas où 16 serait une limite bien trop faible.

4 Tests

4.1 Partie I

Pour tester notre première implémentation des threads utilisateur, nous avons choisi de créer des tests les plus concis possibles afin de pouvoir trouver plus rapidement les bugs qui pourraient y être liés. Nous avons donc commencé par tester les appels systèmes *SC_ThreadCreate* et *SC_ThreadExit* avec le test *test/makethreads*. Même si celui-ci a changé au cours du développement des threads utilisateurs, on peut voir que son intérêt principal était de voir si l'on arrivait a exécuté du code dans un thread (qui n'était pas le thread *main*).

4.2 Partie II

Les tests sont devenus beaucoup plus important à partir de cette partie, car le fait d'avoir plusieurs threads en même temps a provoqué des bugs plus difficiles à identifier. L'un des enjeux de l'exécution de plusieurs threads simultanément étant de pouvoir utiliser les fonctions que nous avions implémentées dans le *TD1*, il a donc fallu les adapter pour les utiliser. Nous avons donc commencé et testé l'adaptation sur *PutChar* et *GetChar* sans problème avec les programmes *test/threadlock* et *test/threadgetchar*. En effet celui-ci permet de mettre des appels en concurrence sur ces fonctions et donc de vérifier leur synchronisation. Ce programme permet également de tester notre implémentation de la classe *Lock* dans *synch.cc* car nous avons implémenté la synchronisation de ces fonctions avec ceux-ci. Ces programmes nous on permit de trouver un bug de synchronisation. Nous n'avions pas au préalable protégé les fonctions *GetString* et *PutString*, lorsque nous avons voulu remplacer les appels à *PutChar* et *GetChar* par *PutString* et *GetString* dans les deux programmes de tests précédents, nous avons commencé à voir des appels qui se mélangeaient (nous avons donc ajouté une synchronisation à ces deux fonctions). Nous avons donc ajouté le test *test/threadstring* qui permet de tester la synchronisation de ces

fonctions. De plus, pour vérifier le bon fonctionnement de notre allocation de pile, nous avons ajouté le test *test/threadloop*. Celui-ci permet de vérifier que les variables sur la pile sont bien locales à chaque thread (et que donc les piles sont bien allouées) grâce à une boucle *for* qui est exécutée en parallèle sur 3 threads. C'est également dans ce test que nous avons découvert le bug du *printf* dans les boucles (nous avons donc utilisé un mutex et des appels successifs à *PutChar*, *PutString* et *PutInt* pour simuler le *printf*). Enfin, pour vérifier notre système de création de thread et d'allocation des piles, nous avons créé le test *test/maxthreadcreate*. Ce test doit créer une série de 5 threads, mais *AddrSpace* n'a de la place que pour 4 pile par défaut. Donc si tous les threads se créaient avant qu'un ait pu finir (ce qu'il est censé se passer avec les arguments de test par défaut), au moins 1 thread ne doit pas réussir à se créer. Cependant, le programme ne se fini pas brutalement et se contente de remonter la valeur -1 à l'appel *ThreadCreate* pour signaler que le thread n'est pas créé.

4.3 Partie III

Cette partie n'a pas de test en particulier, car nous avons retiré les appels à *ThreadExit* dans toutes nos fonctions de test. Comme tous nos tests s'exécutent de la même manière sans cet appel, nous considérons que chacun est un test de la terminaison automatique.

4.4 Partie IV

Pour tester les sémaphores utilisateurs, nous avons créé deux programmes. Le premier, *semaphoresyscalls* permet de tester les différentes fonctions et appels systèmes que nous avons implémentés. Le second correspond au programme producteur-consommateur demandé (basé sur l'exemple présent dans le cours). Cependant, nous avons dû apporter quelques modifications car nous n'avons pas accès aux files côté utilisateur. Nous avons donc détaillé notre implémentation dans le commentaire de ce test.