Nachos: Rapport final

Borne, Duquennoy, Duverney, Isnel

1 Fonctionnalités

Environnement utilisateur multi-processus avec espaces d'adressages virtuels. Processus utilisateur multi-threadés avec primitives de synchronisation (Mutex, Semaphores, Variables-conditions). Gestion des entrées-sorties (`GetInt`, `PutInt`, `GetChar`, `PutChar`, `GetString`, `Putstring`). Système de fichier. Protocole réseau sans connexion avec envoi fiable de données. Utilisation interactive via un Shell (`exec`, `cd`, `ls`, `mkdir`, `rm`, `touch`, `cat`, `messagerie réseau instantanée` ...).

2 Appels système

2.1 Entrées-Sorties

`void PutChar(char c)`

- Sémantique : Écrit le caractère `c` sur la sortie standard.

`void PutString(const char *s)`

- Sémantique : Écrit la chaîne de caractères lue à l'adresse `s` sur la sortie standard.
- Préconditions : La chaîne doit se terminer par un caractère `'\0'`.

`char GetChar()`

- Sémantique : Lit un caractère depuis l'entrée standard et retourne le caractère lu.

`void GetString(char *s, int n)`

- Sémantique : Lit une chaîne de caractères de longueur maximale égale à `n` depuis l'entrée standard et l'écrit à l'adresse `s`.
- Préconditions : L'adresse `s` est valide (espace suffisant).

`void PutInt(int n)`

- Sémantique : Écrit l'entier `n` sur la sortie standard.

`void GetInt(int *n)`

- Sémantique : Lit un entier depuis l'entrée standard et l'écrit à l'adresse `n`.
- Préconditions : L'adresse `n` est valide.

2.2 Processus et Threads

`int ForkExec(char * fileName)`

- Sémantique : Crée un nouveau processus qui exécute le fichier dont le nom est fourni en paramètre
- Préconditions : "fileName" est le nom d'un fichier exécutable au format `noff`.
- Valeur de retour : retourne l'identificateur du thread/processus créé, -1 si une erreur s'est produite lors de la création du thread.

`int UserThreadCreate(void f(void* arg), void* arg)`

- Spécifications : Prends en paramètres un pointeur de fonction "f" ne retournant pas de valeur et un pointeur "arg" vers le ou les paramètres de la fonction "f".
- Sémantique : Crée un nouveau thread utilisateur qui exécute la fonction `f(arg)`.
- Préconditions : Le système doit disposer d'une quantité de mémoire suffisante pour allouer la pile du thread à créer.
- Valeur de retour : retourne l'identificateur du thread créé, -1 si une erreur s'est produite lors de la création du thread.

`void UserThreadExit()`

- Sémantique : Termine l'exécution du thread courant.

`int UserThreadJoin(int tid)`

- Sémantique : Attend la terminaison du thread d'identifiant "tid", renvoie -1 si le thread est déjà terminé, 0 sinon.
- Préconditions : "tid" est un identifiant de thread valide, initialisé dans le processus courant à l'aide d'un appel à `UserThreadCreate`. Le thread courant n'a pas déjà fait `join(tid)`.

2.3 Synchronisation

`Mutex_t MutexCreate()`

- Sémantique : Initialise un mutex.
- Valeur de retour : Un identifiant de type `Mutex_t` pour le mutex.

`void MutexLock(Mutex_t mutexId)`

- Sémantique : Acquiert le mutex d'identifiant "mutexId". Si "mutexId" est déverrouillé, il devient verrouillé et possédé par le thread appelant. Si le mutex d'identifiant "mutexId" est déjà verrouillé par un autre thread, le thread courant est suspendu jusqu'à ce que "mutexId" soit déverrouillé.
- Pré-Condition : "mutexId" est un identifiant de mutex valide, initialisé dans le processus courant par la méthode `MutexCreate`.

`void MutexUnlock(Mutex_t mutexId)`

- Sémantique : Relâche le mutex d'identifiant "mutexId".
- Précondition : "mutexId" est un identifiant de mutex valide, initialisé dans le processus courant par la méthode `MutexCreate`. "mutexId" est verrouillé.

`MutexDestroy(Mutex_t mutexId)`

- Sémantique : Détruit le mutex "mutexId".
- Préconditions : "mutexId" est un identifiant de mutex valide, initialisé dans le processus courant par la méthode `MutexCreate`. Le verrou est relâché. Détruire un verrou non relâché mène à un comportement non déterminé.

`Sem_t SemCreate(int initialValue)`

- Sémantique : Initialise une sémaphore avec la valeur "initialValue".
- Valeur de retour : Un identifiant de type `Sem_t` pour la variable-condition.

`void SemWait(Sem_t semaphoreId)`

- Sémantique : Le thread courant attend que la sémaphore ait une valeur supérieure à 0 et la décrémente.
- Précondition : "semaphoreId" est un identifiant de sémaphore valide, initialisé dans le processus appellant par la méthode `SemCreate`.

`void SemPost(Sem_t semaphoreId)`

- Sémantique : Incrémente la valeur de la sémaphore, réveille un thread en attente de cette sémaphore si besoin.
- Préconditions : "semaphoreId" est un identifiant de semaphore valide, initialisé dans le processus courant par la méthode `SemCreate`.

`void SemDestroy(Sem_t semaphoreId)`

- Sémantique : Libère les ressources associées à la sémaphore.
- Préconditions : "semaphoreId" est un identifiant de semaphore valide, initialisé dans le processus courant par la méthode `SemCreate`. Aucun thread ne doit être en attente de la sémaphore.

`Cond_t CondCreate()`

- Sémantique : Initialise une variable-condition.
- Valeur de retour : Un identifiant de type `Cond_t` pour la variable-condition.

`void CondWait(Cond_t condId, Mutex_t mutedId)`

- Sémantique : Met le thread courant en sommeil dans la file d'attente associée à "condId" et relâche le verrou "mutexId".
- Préconditions : "mutexId" et "condId" sont des identifiants de mutex et variables-conditions valides, initialisés dans le processus courant par les méthodes `MutexCreate` et `CondCreate`. "mutexId" est verrouillé.

`void CondSignal(Cond_t condId)`

- Sémantique : Réveille un thread en sommeil dans la file d'attente associée à la variable-condition "condId". Si aucun thread n'est présent dans la liste, le signal est perdu.
- Préconditions : "contId" est un identifiant de variable-condition valide, initialisé dans le processus courant par la méthode `CondCreate`.

`void CondBroadCast(Cond_t condId)`

- Sémantique : Réveille tous les threads en sommeil dans la file d'attente associée à la variable-condition "condId".
- Préconditions : "contId" est un identifiant de variable-condition valide, initialisé dans le processus courant par la méthode `CondCreate`.

`void CondDestroy(Cond_t condId)`

- Sémantique : Libère les ressources associées à la variable condition "condId".
- Préconditions : "contId" est un identifiant de variable-condition valide, initialisé dans le processus courant par la méthode `CondCreate`. Aucun thread n'est en attente dans la file associée à la variable.

2.4 Système de fichier

`void Create (char *name, int initialSize)`

- Sémantique : Crée un fichier de nom "name" et de taille "initialSize".

`OpenFileId Open(char *name)`

- Sémantique : Ouvre le fichier dont le nom est "name".
- Valeur de retour : Retourne un descripteur de fichier de type `OpenFileId` permettant de lire et écrire dans le fichier ou -1 si l'ouverture à échoué.

`int Write (char *buffer, int size, OpenFileId id)`

- Sémantique : Écrit "size" octet(s) depuis le fichier dont le descripteur est "id" dans le buffer "buffer".
- Préconditions : Le descripteur "id" doit être valide (fichier ouvert), initialisé dans le thread courant par la méthode `Open`. Tenter d'écrire dans un fichier non initialisé par open resulte en un comportement non spécifié.
- Valeur de retour : Nombre d'octets écrits dans le fichier.

`int Read(char *buffer, int size, OpenFileId id)`

- Sémantique : Lit "size" octets depuis le fichier dont le descripteur est "id" dans le buffer dont l'adresse est "buffer". Si le fichier contient moins de "size" octets on lit tout les octets disponibles.
- Préconditions : Le descripteur doit être valide (fichier ouvert), initialisé dans le thread courant par la méthode `Open`. Tenter de lire dans un fichier non initialisé par `Open` résulte en un comportement non spécifié.
- Valeur de retour : Nombre d'octets lus.

`void Close(OpenFileId id)`

- Sémantique : Ferme le fichier dont le descripteur est "id".

- Préconditions : Le descripteur doit être valide (fichier ouvert), initialisé dans le thread courant par la méthode `Open`. Tenter de fermer un fichier non initialisé par `Open` résulte en un comportement non spécifié.

`void CreateDirectory(char * name)`

- Sémantique : Crée un répertoire dans le système de fichier Nachos, de nom "name" passé en paramètre.

`void ChangeDirectoryPath(char * name)`

- Sémantique : Change le répertoire courant vers le répertoire de nom "name".
- Exemple : `ChangeDirectoryPath("./Dossier1/Dossier2")`

`void ListDirectory(char * name)`

- Sémantique : Liste tout les fichiers et documents du répertoire dont le nom "name" est passé en paramètre.
- Exemple : `ListDirectory("./Dossier1/Dossier2")`

`int Remove(char * name)`

- Sémantique : Supprime le fichier ou répertoire dont le nom "name" est passé en paramètre.
- Valeur de retour : 1 si la suppression a réussi, 0 sinon.
- Exemple : `Remove("./Dossier1/Dossier2")`

2.5 Réseau

`void SendMessage(int addressDesti, int boxTo, int boxFrom, char * data)`

- Sémantique : Envoi du message "data" depuis la boite "boxfrom" vers la machine d'adresse "addressDesti" dans la boite "boxTo".
- Préconditions : La machine "addressDesti" doit être prête à recevoir des messages, i.e. avoir exécuté `ReceiveMessage`, les numéros de boites "boxTo" et "boxFrom" sont compris entre 0 et 9.

`void ReceiveMessage(char * data, int box)`

- Sémantique : Initialise la réception d'un message depuis la boite "box". Les données reçues sont stockées à l'adresse "data".
- Préconditions : Le numéro de boite "box".

3 Implémentation

3.1 Processus et Threads

`Fichiers: userprog/userthread.cc, userprog/userprocess.cc, userprog/addrspace.cc, thread/thread.cc`

Nous modélisons un processus par un objet `AddrSpace`. Les fonctionnalités rendues par un objet `AddrSpace` dépassent la simple gestion de la mémoire puisque l'on trouve encapsulé au sein de cette classe, les méthodes relatives, entre autres, à la restauration et la sauvegarde de l'état du processeur de la machine `MIPS`. Nous avons décidé d'étendre la sémantique de l'objet `addrSpace` en ajoutant une liste des threads actifs dans l'espace d'adressage, ainsi que les informations relatives aux éventuels appels à la méthode `join` entre ces threads.
Définition. On appellera *espace d'adressage* une entité décrivant un espace mémoire, l'état de la machine `MIPS` et les informations relatives aux fils d'exécution actifs dans cet espace.

Afin de faire le lien entre les différents threads d'un même processus/espace d'adressage nous donnons à chaque objet `thread` une référence vers un objet `Addrspace`.

Remarque. Lors de notre réflexion sur l'implémentation des processus utilisateur, il nous semblait plus cohérent de déléguer à `addrspace` les méthodes de gestion de la mémoire et de créer une classe `processus` regroupant d'un côté un espace d'adressage et de l'autre les informations relatives aux fils d'exécutions vivant dans cet espace. Pour un petit gain de cohérence nous avons introduit une trop grande complexité de mise en oeuvre. Nous avons donc abandonné ce modèle au profit de celui décrit plus haut.

Piles des threads

Afin d'accueillir les piles de nouveaux threads on divise un espace d'adressage `addrSpace` en blocs de pages, de taille `NumPagesPerStack`. Dans chaque espace on mémorise l'état courant des emplacements de piles grâce à la bitmap `stackBitmap`.

Thread Join

Au sein d'un processus, un thread peut utiliser la méthode `UserThreadJoin` pour attendre la terminaison d'un autre thread. On maintient dans chaque `addrSpace` une structure `joinMap`. La map associe à un `tid` une liste de threads.

Lorsqu'un thread $t1$ fait un `join` sur un thread $t2$ on ajoute dans `joinMap` l'association $(t2, [t1])$. Cette association signifie que $t2$ est attendu par $t1$. Le

thread $t1$ se met alors en sommeil et attends $t2$. Si $t3$ fait un `join` sur $t2$ la `joinMap` se retrouve dans l'état $(t2, [t1, t3])$ et $t3$ est mis en sommeil. Lorsque $t2$ termine son exécution il place $t1$ et $t3$ dans la `readylist` du scheduler.

De plus, on maintient chaque `addrSpace` une liste `threadList` des threads actifs de l'espace d'adressage. Un thread $t1$ ne peut faire un join que vers un thread $t2$ actif. Un thread peut être attendu par plusieurs threads distincts.

Création d'un thread

La création d'un thread utilisateur s'effectue par un appel à `UserThreadCreate` qui déclenche un passage en mode noyau via l'instruction `syscall`. Le gestionnaire d'exception implémenté dans la méthode `ExceptionHandler` de la classe `exception.cc` appelle alors la méthode `do_UserThreadCreate()`. L'initialisation du nouveau fil d'exécution s'effectue en trois étapes.

1. Dans `do_UserThreadCreate` :

   On crée un nouvel objet thread t (thread noyau propulseur), destiné à configurer et brancher la machine `MIPS` sur l'exécution d'une nouvelle fonction f au sein de l'espace d'adressage du thread/processus courant. On commence par récupérer la fonction utilisateur f à exécuter et ses arguments arg dans les registres 4 et 5 de la machine `MIPS`. On appelle ensuite la méthode `t->Fork(StartUserthead, (f, arg))`.

2. Dans `Fork` : On initialise la "partie noyau" du thread propulseur t de manière à ce qu'il exécute la méthode `StartUserThread(f,arg)`. On affecte à t le même espace d'adressage que le thread courant. On met à jour la liste des threads de l'espace d'adressage, on cherche un emplacement libre dans l'espace d'adressage pour la pile du nouveau thread t. On place ensuite le thread t dans la `readyList` du scheduler.

3. Lorsque t devient le thread courant il exécute la méthode

   `StartUserThread(f,arg)` qui se charge de configurer les registres de la machine `MIPS` pour l'exécution de la fonction utilisateur `f(arg)` et de lancer la machine avec `machine->Run()`.

Création d'un processus

La procédure de création d'un processus est similaire à celle d'un thread utilisateur, à la différence que l'on doit lire un exécutable au format `noff` pour initialiser un nouvel espace d'adressage et affecter cet espace au thread propulseur avant l' appel à `Fork` dans la méthode `do_UserProcessCreate`.

Terminaison d'un thread et d'un processus

Les threads d'un processus ont une vie (et une mort) indépendante, sans aucune hiérarchie entre eux. Un processus se termine en même temps que son dernier fil d'exécution. L'absence de hiérarchie entre threads à l'intérieur d'un processus et entre processus simplifie l'implémentation des méthodes `do_UserThreadCreate` et `do_UserThreadExit`.

La terminaison d'un thread utilisateur se fait par l'appel système `User ThreadFinish` qui appelle `do_UserThreadExit`. Le thread courant consulte la `joinMap`, réveille les thread en attente. Enfin on appelle la méthode `finish` dans laquelle on met à jour la `stackBitmap`, la liste des threads de l'espace d'adressage, on décrémente le compteur des threads actifs sur le système. Le thread courant devient alors `threadTobeDestroyed` (ou éteint la machine si il était le dernier thread actif).

Au prochain appel de la méthode `run`, si le thread à détruire était le dernier thread d'un espace d'adressage, le scheduler appelle `do_UserThreadExit` pour libérer les ressources et la mémoire associée au processus et détruit le thread propulseur.

Terminaison automatique des threads

Dans `Start.S`, lors de l'appel système `UserThreadCreate`, on place dans le registre 6 de la machine `MIPS` l'adresse de l'instruction `UserThreadExit`. Lors de la création du nouveau thread au niveau noyau avec `do_UserThreadcreate` on récupère cette adresse depuis `r6` et au moment de `Fork`, avant de brancher la machine `MIPS` sur le nouveau thread utilisateur, on place cette adresse dans le registre `RetAddrReg`. Ainsi au moment de terminer son exécution le `PC` sera branché sur `UserThreadExit`.

Mémoire virtuelle

La virtualisation de la mémoire des processus utilisateur est mise en oeuvre par une association entre numéro de page en mémoire virtuelle et un numéro de page physique. Au moment de la création d'un espace d'adressage, l'allocation des pages physiques est réalisée par la méthode `GetEmptyframeRandom()` de l'objet `FrameProvider`. Dans cette méthode on construit, à l'aide de la `bitmap` de l'objet `FrameProvider` une table temporaire des numéros de cadres de pages libres. On choisit ensuite aléatoirement un numéro de cadre dans cette table. La construction de cette table est coûteuse en temps d'exécution mais facilite la mise en oeuvre du tirage aléatoire.

3.2 Synchronisation

La gestion des primitives de synchronisation au niveau utilisateur s'appuie sur les structures de données noyau suivantes que l'on trouvera dans la classe `system.cc` :

- `unsigned int mutexCounter` : Nombre total de mutex créés, sert d'indentifiant lors de l'initialisation d'un mutex au niveau utilisateur.
- `std::map<int,Lock * > * mutexMap` : Association entre un identifiant de mutex et un objet `Lock`.
- `unsigned int semCounter` : Nombre total de sémaphores créées, sert d'identifiant lors de l'initialisation d'une sémaphores au niveau utilisateur.
- `std::map<int,Semaphore * > * semMap` : Association entre un identifiant de sémaphore et un objet `Semaphore`.
- `unsigned int condCounter` : Nombre total de variables conditions crées, sert d'identifiant lors de l'initialisation d'une variable condition utilisateur.
- `std::map<int,Condition * > * condMap` : Association entre l'identifiant d'une variable condition et un objet `Condition`.

nous avons ajouté au fichier `usersynch.cc` un ensemble de méthodes pour la création des objets mutex, sémaphores, variables-condition et la mise à jour des structures au niveau noyau servant de support à la synchronisation dans les processus utilisateurs.

3.3 Réseau

Nous avons mis en place un protocole simple de transmission fiable de données (sans perte de d'information), qui s'inspire du mécanisme d'acquittement cumulatif de `TCP` et de la notion de flux d'octet.

Définition. On appelle message, une séquence d'octets à transmettre sur le réseau.

L'envoi d'un message passe par l'envoi d'un ou plusieurs objets `Mail` que l'on peut assimiler à un segment `TCP`. Un mail est constitué d'un en-tête et d'un tableau d'octets contenant les données à envoyer. On distingue trois types de mail :

- Données
- Acquittement
- Fin de message

La fiabilité de notre protocole, tout comme `TCP` s'appuie sur l'utilisation de numéros de séquences et de numéros d'acquittements.

Numéro de séquence

Supposons que l'on initie un flux d'octet bidirectionnel entre un serveur `A` et un client `B`. Pour chaque direction de flux, tout octet du flux est identifié par un numéro de séquence unique. Lorsque l'on envoie un mail, on joint dans l'entête du mail un numéro de séquence correspondant au premier octet des données envoyées dans ce mail. Le numéro de séquence initial d'un flux de donnée est 0.

Numéro d'acquittement

La fiabilité de la transmission repose sur le fait que, pour chaque mail envoyé par A, B prévient A qu'il a bien reçu le mail en lui renvoyant un mail d'acquittement. Un mail d'acquittement est un mail particulier, sans données, servant uniquement à transmettre un numéro d'acquittement correspondant au numéro de séquence du prochain octet que B s'attend à recevoir. Avec le principe d'acquittement cumulatif, `A` n'a pas besoin de recevoir les acquittements de tous les paquets qu'il envoie. Si on envoie trois paquets p1, p2, p3 à `B` et que `B` acquitte p3, alors si `A` reçoit l'acquittement il sait que `B` a bien reçu p1, p2 et p3.

Timer de réemission

Chaque mail, avant d'être envoyé, est encapsulé dans un paquet au niveau de la couche réseau (semblable à IP). A chaque envoi de paquet (à l'exception des paquet d'acquittements) on déclenche un timer. A la "sonnerie" du timer, si on a pas reçu d'acquittement, on réémet le paquet considéré perdu. Il se peut aussi que les données arrivent mais que l'acquittement se perde, le résultat est le même.

Mise à jour des champs ACK et SEQ

Lors d'échanges de paquets entre `A` et `B` on incrémente les valeurs des champs séquence et acquittement. Pour simplifier notre protocole on exige que les envois de données se fassent dans l'ordre. Le numéro d'acquittement correspond au dernier octet reçu + 1. Le numéro de séquence correspond à l'indice du premier octet du paquet envoyé. L'émetteur d'un acquittement ne sait pas si celui-ci est bien arrivé à destination (on acquitte pas un acquittement).

Envoi et réception messages de tailles variables

Lors de l'envoi d'un message dépassant `MaxMailSize` on découpe le message en plusieurs mails. L'émetteur termine l'envoi du message par un mail de type "Fin de message" contenant pour seule donnée la chaîne de caractères "FIN". Une fois ce dernier mail acquitté, A est sûr que le message est bien arrivé.

4 Système de Fichiers

Nous avons ajouté à l'objet `Filesys` un tableau `fileDescriptor* fileOpened[NBFILEOPENED]` qui mémorise les descripteurs des fichiers ouverts dans le système de fichiers. Ce tableau est utilisé pour garantir l'accès exclusif d'un fichier à un unique thread sur tout le système. Lorsqu'un fichier est ouvert depuis un thread utilisateur on place, au niveau noyau, le descripteur du fichier "f" dans la première case libre du tableau `fileOpened` et l'appel à `Open` retourne l'indice de cette case au thread. On s'assure ensuite que le descripteur d'un même fichier ne se trouve jamais deux fois dans le tableau `fileopened`.

Aussi, nous avons ajouté un attribut `bool isDirectory` à la classe `Directory Entry`. Cet attribut nous permet de discerner dans un dossier, quelle entrée est un fichier ou un répertoire.

5 Organisation

Pour la quasi-totalité des étapes 1 à 4, nous avons travaillé sur un ordinateur commun. Lors des étapes 5 et 6 nous nous sommes scindés en trois sous-groupes, restant malgré tout en communication constante sur les décisions d'implémentation. En ce qui concerne la répartition temporelle des tâches, nous avons essayé de suivre le planning prévisionnel proposé. Nous avons rencontré des difficultés lors de l'étape 4 et avons sans doute hésité trop longtemps à renoncer à un "mauvais" choix de conception évoqué dans la section "processus et threads" de ce rapport. Une fois la décision prise de revenir en arrière nous avons pu remettre notre projet sur une bonne voie. La forte implication de tous les membres du groupe dans le projet nous a permis d'échanger de manière constructive sur les choix d'implémentation, de s'assurer que nous avions tous une compréhension et une vision homogène du fonctionnement de notre code et du système nachos. Nous avons pu ainsi adopter un point de vue critique vis à vis du code produit, n'hésitant pas à discuter en détail des points qui ne nous semblaient pas clairs, à remettre en question des choix, suggérer des corrections de code, de style d'écriture. Nous n'avons éprouvé aucunes difficultés à travailler en groupe, et n'avons rencontré, c'est assez rare pour le préciser, aucune ombre de conflit relationnel lors du déroulement de ce projet.

6 Pistes d'améliorations

Nous avons, d'une manière générale, pris le parti de choisir les conceptions les plus simples lors de la mise en oeuvre des fonctionnalités de notre système. Cela nous à permis d'aboutir à un code fonctionnel, simple à comprendre et à maintenir et de nous familiariser avec nachos. Nous pourrions à présent raffiner

nos implémentations et proposer des solutions plus perfectionnées/performantes. Nous estimons que dans l'état actuel du projet nous pourrions rapidement mettre en oeuvre les amélioration suivantes : Bibliothèque d'entrées-sorties bufferisées, allocateur mémoire, nouvelles politiques pour le scheduler, ouverture concurrente de fichiers, taille variables des fichiers. Nous ne pouvons pas encore exécuter de manière interactive des programmes utilisateurs depuis notre invite de commande. Il nous faudrait pour cela implémenter un minimum de communication inter-processus (Signaux, waitPid) ce qui serait aussi envisageable à court terme.

7 Conclusion

Nous avons le sentiment unanime d'avoir partagé une excellente expérience de travail collaboratif. Nous avons pris plaisir à confronter nos connaissances à un cas d'utilisation pratique et sommes très heureux d'avoir pu participer à ce projet. Nous tenons à remercier Vincent Danjean et Vania Marangozova pour leur support et leur enseignement.

8 Annexe

8.1 Tests

Entrées-Sorties

- `Putchar_0` : Écriture du caractère `'a'` sur la sortie standard.
- `Putchar_1` : Écriture des caractères `'a'` et `'b'` sur la sortie standard.
- `Putchar_2` : Écriture de multiples caractères sur la sortie standard.
- `PutInt_0` : Écriture de l'entier `10` sur la sortie standard.
- `PutInt_1` : Écriture de l'entier `0` et de l'entier `1` sur la sortie standard.
- `GetInt_0` : Lecture d'un entier depuis l'entrée standard.
- `GetInt_PutInt_0` : Lecture d'un entier depuis l'entrée standard. Écriture de cet entier sur la sortie standard.
- `GetString_0` : Lecture d'une chaîne de moins de `20` caractères depuis l'entrée standard, affichage de cette chaîne sur la sortie standard.
- `PutString_0` : Affichage de la chaîne de caractère `"ABCDEFGHIJklmnop qrstuvwxyz"`.
- `PutString_1` : Affichage de la chaîne `"ABCDEFGH"` et `"ijklmnopqrstuvwxyz"`.
- `PutString_2` : Affichage d'une chaîne de taille supérieure à la constante `MAX_STRING_SIZE(=100)` définie dans `system.h`.

UserThreads

- `UserThreadcreate_0` : Lancement de `N` threads utilisateurs qui affichent chacun un entier passé en paramètre lors de leur création.
- `Userthreadcreate_1` : Lancement de deux threads utilisateur. Le premier affiche `"X"` puis le caractère `'a'` passé en paramètre. Le second affiche `"Y"` puis le caractère `'b'` passé en paramètre.
- `Userthreadcreate_2` : Lancement d'un nombre de thread trop important pour l'espace mémoire. Le programme affiche l'identifiant des threads créées.

Entrées-Sorties Multithread

Avec les tests suivants, on vérifie le bon fonctionnement de synchconsole.

- `MultithreadGetChar_0` : Création de plusieurs threads exécutant `GetChar` puis `Putchar`. Remarque : A l'exécution, lorsqu'un thread s'exécute il se bloque sur l'instruction `GetChar()` et attends une entrée utilisateur. L'utilisateur entre un caractère `'c'` et `'\n'` pour terminer son entrée. Suite à l'appel de `GetChar()`, le caractère `'\n'` est toujours présent dans l'entrée standard. Aussi, le thread suivant exécutant `GetChar()` récupère et affiche `'\n'`.
- `MultithreadGetInt_0` : Création de plusieurs threads exécutant `GetInt` puis `PutInt` puis `PutInt` pour afficher l'entier saisi.
- `MultithreadGetString_0` : Création de plusieurs threads exécutant `get String` puis `PutString`.

On reprend l'ensemble des tests précédents sur les opérations d'entrées-sorties en rendant l'exécution de chaque thread séquentielle par l'utilisation d'un mutex ou d'une sémaphore.

Lecteur-rédacteurs

Afin de vérifier le bon fonctionnement des variables-conditions et mutex au niveau utilisateur, nous avons implémenté une solution au problème des lecteurs-rédacteurs.

ForkExec

- `ForkExec_0` : Crée un nouveau processus qui exécute le programme `PutInt_0`.
- `ForkExec_1` : Crée un nouveau processus qui exécute le programme `PutChar_0`.
- `ForkExec_2` : Crée un nouveau processus qui exécute le programme `ForkExec_0`.

- `ForkExec_3` : Crée un nouveau processus qui exécute le programme `ForkExec_1`.
- `ForkExec_MultiThread_0` : Crée un nouveau processus qui exécute le programme `MultiThreadPutInt_Mutex_0`.
- `ForkExec_MultiThread_1` : Crée un nouveau processus qui exécute le programme `ForkExec_MultiThread_0`.

Nous vérifions dans gdb les changements d'espace d'adressage, la libération de la mémoire lors de la terminaison d'un processus utilisateur.

Shell

Nous avons implémenté un petit programme shell qui permet une utilisation interactive de notre système. Les commandes suivantes sont implémentées :

- `exec` : prends en argument un nom de fichier de programme exécutable au format noff, crée un nouveau processus utilisateur qui exécute le programme.
- `cd` : change le répertoire courant vers un nouveau répertoire.
- `ls` : liste le contenu du répertoire courant si il n'y a pas d'argument, sinon liste le contenu du répertoire en argument.
- `mkdir` : crée un répertoire dont le nom est fourni en argument.
- `cat` : affiche le contenu du fichier dont le nom est fourni en argument.
- `rm` : supprime un répertoire ou un fichier fourni en argument.
- `touch` : crée un nouveau fichier vide dont le nom est fourni en argument.
- `chat` : initie une messagerie instantanée avec une autre machine nachos connectée au réseau.

Nous avons testé le fonctionnement de notre système de fichier avec ce programme. nous avons pu établir une messagerie réseau instantanée entre deux machines pour vérifier le fonctionnement de notre protocole de transfert de données. Enfin nous avons pu exécuter les programmes de test précédent depuis le shell.