

H-NACHOS : LE RAPPORT

R. Soumard, C. Semanaz, W. Ayari, J. Neola

Project Système - 2021



Manuel de spécification	2
Appels système	
Tests utilisateur	3
Entrées et sorties	
Threads	
Gestion de fichiers	
Choix d'implémentation	6
Entrées/sorties	
Threads	
Mémoire	
Processus	
Système de fichier	
Réseau	
Synchronisation	
Organisation	10
Planning	
Bug connus	11

H-NACHOS : SPÉCIFICITÉS ET FONCTIONNALITÉS

H-NachOS est un système d'exploitation minimaliste réalisé en projet système durant l'année 2020-2021. Au cours du développement, nous y avons apportés les fonctionnalités suivantes: Fonction basiques d'entrées/sorties, système de multi-threading utilisateur, gestion de multiples processus, adressage statique de la mémoire, et mécaniques d'attente. Nous y avons en plus ajouté un système d'attente limité, afin de synchroniser la partie réseau à posteriori. Avec davantage de temps, il était prévu d'ajouter un système de communication fiable en réseau, s'appuyant sur TCP/IP, rendre la lecture des fichiers concurrente au sein du système de fichier, et corriger certains bugs restants.

Manuel de spécification

APPELS SYSTÈME

Cette section détaille les appels système disponibles aux programmes utilisateurs. Ceux-ci sont accessibles avec l'import de `"syscall.h"` en début de fichier.

Entrées et sorties

`char` GetChar()

Lit un caractère sur l'entrée standard. Si aucun caractère n'est présent sur l'entrée, l'appel est bloquant. Renvoie une variable de type `char` représentant le caractère lu sur l'entrée.

`void` PutChar(`char` c)

Affiche le caractère c sur la sortie standard. PutChar ne gère que les caractères encodés sur 1 octet.

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

Lit size (au maximum 63) octets sur l'entrée standard et les stocke dans le buffer s passé en argument. La chaîne lue est terminée par un caractère `'\0'`. La lecture s'arrête si on rencontre un EOF ou un retour à la ligne. Si size dépasse 63 octets, la chaîne ne sera pas terminée correctement. Si elle dépasse 64 octets de surcroît, elle sera tronquée.

`void` PutString(`char` *s)

Affiche au plus 63 octets de la chaîne de caractère s sur la sortie standard. La chaîne s devrait se terminer par `'\0'` autrement l'exécution de PutString peut mener à un comportement non-défini (affichage aléatoire). Si la chaîne fait plus de 63 octets, elle sera systématiquement tronquée au 64ème.

`int` GetInt()

Lit un nombre entier compris entre -2^{31} et $2^{31} - 1$ sur l'entrée standard. Si l'entier à lire dépasse la valeur de l'une des bornes, alors l'entier retourné sera la valeur de la borne la plus proche. Retourne l'entier lu.

`void` PutInt(`int` d)

Affiche l'entier signé d sur la sortie standard. La valeur de d peut être comprise entre -2^{31} et $2^{31} - 1$. Si elle dépasse l'une de ces bornes, alors on affiche la valeur de la borne la plus proche.

Threads

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

Crée un thread utilisateur qui exécute la fonction f avec comme argument arg. Retourne le tid du thread créé, ou -1 s'il ne peut être créé. Cette fonction n'a d'effet que si appelée par un processus.

`int` UserThreadJoin(`int` tid)

Provoque l'attente de la terminaison du thread désigné par tid. Un processus peut attendre les threads utilisateur qu'il a créés. Un thread utilisateur peut attendre un autre thread utilisateur partageant le même espace d'adressage, i.e. : deux threads du même parent. En aucun autre cas cet appel système ne devrait être utilisé au risque d'un comportement non-défini. Si un thread demande à attendre un thread déjà terminé ou inexistant, l'attente n'a pas lieu.

`void` UserThreadExit()

Met fin proprement au thread courant. Cette fonction n'a aucun effet si appelée par un processus.

Processus

`int` ForkExec(`char` *filename)

Permet de créer un nouveau processus à partir de l'exécutable de chemin filename. Le processus créé s'exécute parallèlement aux autres processus actifs. Cette fonction n'a pas d'effet si appelée par un thread utilisateur. Retourne l'identifiant du processus créé (pid) ou -1 s'il ne peut être créé.

`void` Halt()

Arrête le **système** immédiatement, sans se préoccuper de la terminaison des processus ou threads encore en cours.

`void End()`

Interrompt le système une fois tous les processus et tous les threads ont correctement terminé. Cette fonction ne doit être appelée que par un processus.

Fichiers

Note : Sur notre système, les noms de fichier sont limités à 25 caractères et le nombre maximal de dossier / fichier par répertoire à 8 (10 en comptant les répertoires courant et parent). Les écritures et les créations de fichier se font dans la limite des capacités du disque dur (soit 2^{17} octets). Les fonctions `Open`, `Create` et `Remove` peuvent toute prendre un chemin relatif en paramètre également.

`void Create (char *name)`

Crée un fichier nommé `name` ou, si `name` se termine par `'/'` un répertoire, dans le répertoire courant s'il n'est ne contient pas déjà plus de 8 fichiers ou dossiers et que cette référence n'existe pas déjà.

`OpenFileId Open (char *name)`

Ouvre le fichier `name`, place le pointeur de lecture du fichier sur son premier octet et renvoie son descripteur, un entier positif représentant son premier secteur. Si `name` s'achève avec le caractère `'/'`, on change de répertoire courant si ce répertoire existe. Si le fichier/répertoire ne peut pas être ouvert, renvoie -1. Si le fichier est déjà ouvert, cet appel attend indéfiniment sa libération (appel bloquant).

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

Écrit le contenu de `buffer` dans le fichier ouvert décrit par `id` sur `size` octets et avance le pointeur de lecture du fichier jusqu'au dernier octet écrit. Si la fin du fichier est atteinte, l'écriture est tronquée. Si `buffer` ou `id` est NULL, `size` est inférieur ou égale à 0 ou que le fichier est fermé ou inexistant, alors l'écriture n'a pas lieu.

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

Lit `size` octets du fichier ouvert décrit par `id`, les écrit dans `buffer` et avance le pointeur de lecture du fichier jusqu'au dernier octet lu. Si la fin du fichier est atteinte, la lecture est tronquée. Si `buffer` ou `id` sont NULL, ou que `size` est inférieur ou égale à 0 on retourne -1. Sinon, on retourne le nombre de caractère lus.

`void Close (OpenFileId id)`

Ferme le fichier ouvert décrit par `id`. Cet appel ne fait rien sur un fichier fermé ou inexistant.

`void Remove(char* name)`

Supprime le fichier `name`, ou supprime le répertoire `name/` s'il existe et est vide.

Tests utilisateur

ENTRÉES ET SORTIES

Entrées système

– getchar

Permet de lire un caractère sur l'entrée standard et de l'écrire sur la sortie standard. Ce test est une boucle, elle cesse lorsqu'on entre le caractère `'q'`. N'importe quel caractère peut être lu : chiffres, caractère spéciaux et lettres. *Semble solide, ne peut pas faire crasher le système*

– getint

Permet de lire un entier inférieur à 2^{31} sur l'entrée standard et de l'afficher sur la sortie standard pour vérifier son exactitude. Si on saisit une lettre ou une chaîne de caractère, on ne lit rien et on retourne simplement 0. Si l'entier est supérieur à 2^{31} , on affiche l'entier 2^{31} et pareillement pour la borne minimale -2^{31} . *Semble solide, ne peut pas faire crasher le système, si une lettre/caractère spécial est entré, écrit 0 + on fait halt*

– getstring

Permet de lire une chaîne de caractères de 10 octets sur l'entrée standard. Cette valeur peut être modifiée dans le fichier test. Cette chaîne sera renvoyée sur la sortie standard pour vérifier la bonne exécution. Notons que certains caractères spéciaux s'écrivent sur deux octets, comme 'ㄅ' ou 'ㄆ'. Si on saisit 10 fois le caractère 'ㄆ', on ne l'écrira que 5 fois. *Semble solide, ne peut pas faire crasher le système.*

Sorties système

– putchar

Permet de tester l'affichage de plusieurs caractères via la fonction `void print(char c, int n)` qui affiche le caractère `c` et les `n - 1` suivants. PutChar ne gère pas les caractères sur deux octets. Cette erreur est attrapée à la compilation. *Semble solide, ne peut pas faire crasher le système.*

– putint

Permet d'afficher un nombre compris entre -2^{31} et 2^{31} . *Semble solide, ne peut pas faire crasher le système.*

– putstring

Permet d'afficher une chaîne de caractères quelconque, d'une taille maximale de 64 octets. *Semble solide, ne peut pas faire crasher le système.*

Console synchronisée

Permet d'afficher les caractères lus dans l'entrée standard les uns à la suite des autres. La boucle cesse lorsque que l'on entre CTR+D dans la console. Ce test est accessible avec la commande `./nachos-final -sc`

THREADS

Threads utilisateur

Les tests de threads utilisateur sont dans les fichiers `thread_*.c` et se différencient par ce qu'ils testent (`fewthreads`, `join`, `automaticend`, `manythreads`). Nous les avons effectués avec la commande `./nachos-final -x testfile`.

Chaque version permet d'effectuer un test différent. L'argument `-rs nb` permet de changer l'ordonnement des threads.

– thread_fewthreads

Permet de tester l'appel successif de plusieurs threads par l'appel système `UserThreadCreate()` (ici 5) en vérifiant que l'ordonnement est bien concurrentiel (avec l'option `-rs`). L'ordre de l'affichage des threads n'est pas garanti par la séquence d'appel.

– thread_join

Permet de tester l'appel successif de deux threads en ajoutant la condition que le thread principal doit attendre la terminaison du premier thread (par la méthode `UserThreadJoin()`) avant de lancer le second. Pour cela, on appelle les threads sur une méthode `countDown()` qui effectue un décompte de 10 à 0. La sortie de chaque thread sera donc clairement définie et elle ne doivent pas être entrelacées quelque soit l'ordonnement.

– thread_automaticend

Ce test est là pour prouver qu'on a pas besoin d'appeler explicitement `End()` à la fin du programme principale. Autrement dit l'utilisateur n'a pas besoin de s'embêter à écrire `End()` à la fin de chacun de ses programmes. Le problème c'est que ce test passe mais certaines ne passent pas (voir 12processes/12threads sans `End()` à la fin.

Permet de vérifier que la fin du `main` déclenche bien une fermeture automatique et propre du système par un appel à `End()`. On vérifie par la même occasion qu'un thread utilisateur se termine automatiquement aussi, sans appel à `UserThreadExit()` de la part de l'utilisateur. Une tentative de création de thread utilisateur par un thread utilisateur et non par un processus a été ajoutée au test. Ce thread n'est pas créé, soit le comportement attendu.

– `thread_manythreads`

Permet de tester le nombre maximum de threads en concurrence. Ce nombre est codé dans la variable `NB_MAX_THREADS` dans le fichier "`addrspace.h`". Ce test tente de créer 100 threads. Lors d'une exécution sérielle, on retrouve bien 10 threads créés à la fin du test (quand `NB_MAX_THREADS = 10`). Cependant, lors d'une exécution séquentielle, ce nombre peut être plus élevé car quand un thread utilisateur finit, sa place est libérée et un autre thread utilisateur peut être créé. Cela dépend du temps d'exécution du `main()` et de la fonction appelée ici `print()`. Plus le code de `main()` est long et plus le code de `print()` est court, plus les threads utilisateur auront le temps de se terminer avant que `main()` ne déclenche la tentative de création d'un nouveau thread et donc plus le nombre de thread utilisateur effectivement créé sera grand.

Threads système

– `process_forkexec`

Permet de tester le fonctionnement de l'appel système `ForkExec()`. Le thread principal appelle deux fois `ForkExec()` avec des programmes effectuant de simples affichages. Ce test permet de vérifier la création d'un nouveau processus ayant un ou plusieurs threads utilisateur, tout en s'assurant de leur bonne cohabitation en mémoire. On vérifie par la même occasion si la synchronisation entre processus est correcte.

– `process_12threads + process_12processes`

Permet de tester la création de 12 processus avec chacun 12 threads. Chaque thread écrit sur la sortie standard l'entier 1. On doit donc obtenir à la fin $12 \times 12 = 144$ fois l'entier 1. Pour passer ce test il a fallu augmenter la taille de la mémoire physique. Nous l'avons fixée à 512. Notons que chaque threads dispose d'une pile de 2 pages. C'est pourquoi il nous faut au minimum 144×2 (le nombre de threads) \times 2 (le nombre de pages pour la pile) = 288 pages uniquement pour les threads. À cela s'ajoutent 11 espaces d'adressage pour le programme `process_12threads` et 1 espace d'adressage pour `process_12processes`.

GESTION DE FICHIERS

Interface système

– `fsconsole`

Nous avons créé une console qui permet la gestion des fichiers et dossiers. Cela permet de se rendre compte de la hiérarchie et d'avoir un visuel clair et direct de l'utilisation des différentes fonctions. L'argument `name` peut être un chemin. Cette console est appelée par la commande `./nachos-final -fc`.

Fonctions de la console

- `new name` : crée un fichier nommé `name` si le dossier n'est pas plein
- `del name` : supprime le fichier nommé `name` s'il existe
- `mkdir name` : crée un nouveau répertoire nommé `name` s'il n'existe pas déjà et que le dossier parent n'est pas plein (contient moins de dix références)

- `rmdir name` : supprime le répertoire `name` s'il existe et est vide
- `cd name` : change le répertoire courant pour le répertoire nommé `name` s'il existe
- `ls` : affiche le contenu du répertoire courant

Appels utilisateur

– `fileys_create`

Permet de tester le bon fonctionnement des cinq appels système : `Create`, `Open`, `Write`, `Read`, et `Close`. Ce test crée un fichier "**f1**" dans le répertoire racine, l'ouvre, écrit dedans puis le ferme. Pour constater cette écriture, il est possible d'utiliser `./nachos-step5 -D`. Ensuite on ouvre puis lit le fichier et on affiche sur la sortie standard le résultat. Il est identique à l'écriture effectuée.

– `fileys_limit`

Permet de tester le bon fonctionnement de la table système des fichiers ouverts. Le fichier crée quinze fichiers distincts, enregistrés dans deux répertoires différents (créés dans le `main()`).

Une boucle de taille 50 tente d'ouvrir un fichier parmi les quinze : on observe que les dix premiers fichiers sont ouverts (`Open()` retourne un descripteur de fichier valide), mais les cinq suivants échouent (les fichiers ont été créés mais il est impossible de les ouvrir, il ne reste plus de place dans la table). L'itération suivante bloque (mais n'échoue pas) : en constatant que le fichier à ouvrir est présent dans la table, le processus attend la libération de la ressource.

– `fileys_directory`

Permet de tester le bon fonction des appels système avec un argument contenant un `'/'` en dernier caractère. L'appel crée deux répertoires : "**rep1**" et "**rep2**", il supprime le premier et crée un fichier dans le second. Pour vérifier qu'il s'est bien déroulé, il est utile d'appeler la console dédiée pour afficher le contenu du répertoire racine et vérifier que le repertoire "**rep2**" est présent et contient le fichier créé.

Réseau

– `test_network_machine_ring.sh`

Un court script permettant de connecter un réseau de nachOS en topologie en anneau. Prends le nombre de machine en paramètre. Affiche un message d'usage si le nombre de paramètre est égal à 0 ou -h. Ce script utilise les paramètre -m et -o de nachOS pour connecter les machines entre elle et fait donc appel à la fonction `MailTest` de nachOS.

Choix d'implémentation

ENTRÉES/SORTIES

Les fonctions d'entrée/sortie sont construites dans la classe `SynchConsole`, elle même située dans les fichiers "**synchconsole.cc**" et "**synchconsole.h**". Cette console synchronisée est construite sur une console initiale plus basique. Des verroux sont placés à chaque fonction pour empêcher une exécution entrelacée des fonctions d'entrées/sorties.

Pour l'utilisateur, chaque écriture ou lecture effectuée sur la console doit apparaitre comme une instruction atomique. Aussi, lorsqu'une écriture ou une lecture est entamée, aucun autre thread ou processus ne doit pouvoir effectuer une écriture ou une lecture. C'est pourquoi, nous utilisons un seul et unique sémaphore avec un seul jeton afin de passer le contenu des fonctions `GetChar()`, `PutChar()`, `GetString()` et `PutString()` en section critique. Le verrou est pris au début de chacune de ces fonctions puis relâché à la fin. Par conséquent, la moindre utilisation de ces fonctions est bloquante.

THREADS

Afin d'implémenter les threads utilisateur au sein de H-NachOS, nous avons commencé par étendre la classe `Thread` fournie en y ajoutant quatre attributs : `id`, `index`, `waitQueue` et `numOfWaitingThreads`. L'`id` est un entier unique généré au moment de la création du thread. tandis que l'`index` nous sert à repérer l'emplacement de notre thread utilisateur au sein d'un tableau de pointeurs de `Thread` implémenté dans la classe `AddrSpace`. Il existe ainsi pour chaque processus un tableau de threads utilisateur dédié. Le thread est inclus dans le tableau au moment de la création d'un thread utilisateur dans `do_UserThreadCreate`. `WaitQueue` est quand à lui un sémaphore initialisé à 0 permettant la synchronisation des threads et `numOfWaiting thread` est un compteur dénotant le nombre de threads en attente sur ce thread-ci.

La classe `Thread` a également été étendue avec une structure `FunctionAndArgs`, nous permettant de passer la fonction et les arguments nécessaires à la création d'un thread utilisateur. Cette implémentation présente l'avantage de nous permettre de passer très facilement l'adresse de la fonction et de ses arguments dans la fonction `StartUserThreadCreate()`.

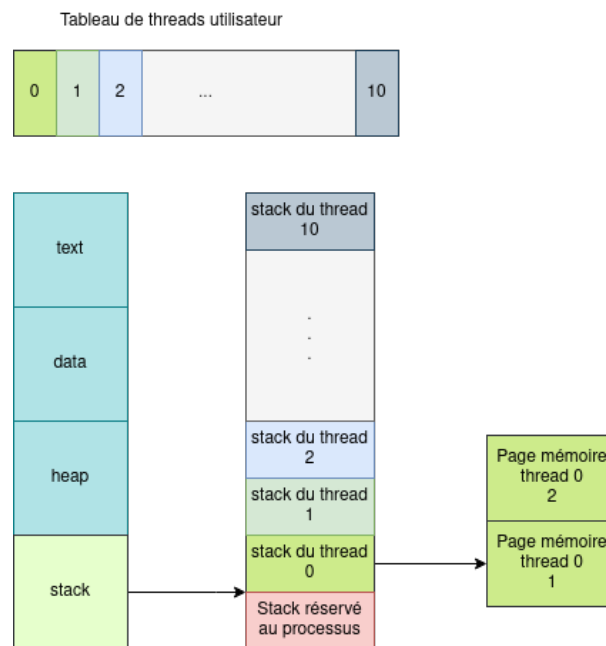


FIGURE 1 – Gestion de la pile

L'enregistrement des threads dans le tableau se fait lors du passage dans la fonction `do_UserThreadCreate` et son effacement dans `do_UserThreadExit`. L'implémentation des threads utilisateur sous forme de tableau nous a permis de limiter le nombre de threads actifs en même temps et de faciliter la gestion de la pile. À chaque index du tableau correspond ainsi une zone de 2 pages de la pile.

On obtient la zone de début de pile de chaque thread utilisateur en ajoutant 1 à l'index puis en multipliant le résultat par $2 \times \text{PageSize}$ qu'on ajoute ensuite à l'adresse de début de la pile du processus. Le décalage de 1 par rapport à l'index est nécessaire afin de garder 2 pages de piles pour le processus principal. La pile est ainsi gérée comme un pool de ressources (réutilisation des mêmes ressources) mais présente le désavantage d'une fragmentation interne. (ressources mémoires non utilisées au sein des slots.)

La taille du tableau est limité par une macro définie au sein du fichier `"addrspace.h"`, nommé `NB_MAX_THREADS`, que nous avons fait varier au besoin au fur et à mesure du développement. La taille de la pile du processus peut ainsi facilement être calculée de la manière suivante : $\text{taille_pile} = 2 \times \text{PageSize} \times (\text{NB_MAX_THREADS} + 1)$. Lorsque le tableau est plein, la création de thread utilisateur supplémentaire est impossible et la fonction `do_UserThreadCreate` renvoie `-1`.

Une synchronisation des threads utilisateur est possible par l'utilisation de l'appel système `UserThreadJoin`, qui prend en paramètre l'`id` du thread utilisateur à attendre. Le processus père peut appeler `UserThreadJoin` sur l'un de ses fils. Un thread utilisateur peut appeler `UserThreadJoin` sur un thread

partageant le même espace d'adressage que lui. Lors d'un appel à `UserThreadJoin`, `waitQueue` est pris et un compteur est incrémenté afin de connaître le nombre de threads en attente. Lors de l'appel à `UserThreadExit()`, le thread libère son sémaphore autant de fois qu'il y a de threads en attente sur ce sémaphore.

Un changement a été effectué dans `"test/start.S"` pour appeler la méthode `End()` à la fin du `main()` du programme. Cela permet un arrêt automatique des fichiers tests. Cela ne fonctionne pas quand on utilise à la suite les méthodes `UserThreadCreate()` et `PutString()` ou `UserThreadJoin()`. *Trouver les limites claires du auto-end*

MÉMOIRE

La politique d'allocation mémoire est principalement encapsulée au sein de la classe `FrameProvider`. Celle-ci se charge d'allouer et de désallouer les pages physiques demandées au sein de la classe `AddrSpace`. Pour cette partie, nous avons implémenté une forme d'allocation statique. Au moment de la création du processus et de son espace d'adressage, leur est alloué un ensemble de pages mémoire physiques. Ces pages mémoire ne sont désallouées qu'à la destruction du processus. `FrameProvider` reposant en interne sur la classe `BitMap`, la politique d'allocation implémentée est celle du *first fit*. (Nous nous servons de `find()` qui se charge de trouver la première frame disponible.)

La classe `AddrSpace` a en plus été étendue afin de pouvoir gérer le tableau de threads utilisateur du processus. Elle contient donc un ensemble de méthodes dédiées au placement et à l'effacement de thread dans ce tableau ainsi qu'une batterie d'autres permettant la recherche d'une place libre, ou celle d'un thread, par id ou par index.

Pour finir nous avons également amélioré en profondeur la classe `AddrSpace` afin de la rendre plus simple à travailler. Pour une description exhaustive des méthodes décrites ci-dessus nous vous laissons vous référer à la documentation interne du code trouvable dans `"userprog/addrspace.cc"`.

PROCESSUS

Notre implémentation des processus passe par une technique relativement similaire à celle des threads utilisateur. Ces derniers reposant sur des threads système, nous nous sommes appuyés sur la classe `Thread` pour les créer. Cette dernière a donc été étendue avec de nouveaux attributs et méthodes. Parmi les ajouts notables, on trouve un identifiant de processus noté `pid`.

Nous avons créé une classe `ProcessTable` afin de représenter et manipuler la table des processus. Cette dernière repose en interne sur la classe `BitMap` que nous avons également légèrement modifié afin de la rendre un peu plus efficace d'un point de vue algorithmique. (ajout d'un attribut `numOfSetBits` pour éviter d'avoir à recalculer constamment le nombre de bits sets.)

La classe `ProcessTable` est hébergée et initialisée au sein de `"system.h"` et `"system.cc"`. Nous avons en effet jugé qu'au contraire des threads utilisateur, la table des processus était un attribut du système lui-même. Afin d'éviter de saturer la mémoire, nous avons limité la taille de ce tableau à l'aide d'une macro `NB_MAX_PROCESS` définie dans `"system.h"`.

`ProcessTable` permet principalement de synchroniser les appels à `End()`, afin d'éviter que la machine ne soit éteinte lorsqu'un processus s'achève alors que les autres n'ont pas terminé. Si un appel à `End()` est fait par un processus, il va soit se terminer et se détruire proprement si d'autres processus sont encore en cours d'exécution, soit éteindre le système si aucun autre processus n'est en cours, le tout grâce au tableau système de processus. À cette synchronisation s'ajoute la synchronisation des threads utilisateurs décrite plus haut. Plus de détails à ce sujet sont visibles dans le fichier `"exception.cc"` dans la fonction `handleEnd()`.

SYSTÈME DE FICHIER

À la gestion des fichiers, nous avons ajouté la gestion des répertoires qui permet d'obtenir un système hiérarchique. Un répertoire est considéré comme un fichier lambda par le système : il y a une entête qui définit le début et la taille du contenu et son contenu (les liens vers les fichiers qu'il contient, au maximum 10 défini par la macro `NumDirEntries` dans `"directory.h"`). La fonction `navigateToPath()`

est appelée dans chaque fonction de création et gestion pour pouvoir gérer un chemin éventuellement donné en argument.

Deux tables ont été définies : la table système des fichiers ouverts et la table thread des fichiers ouverts. La première permet d'avoir plusieurs fichiers ouverts en même temps (jusqu'à dix) pour cela nous avons opté pour une table de structures associant un `OpenFile`, son numéro de secteur et son verrou. Les fonctions `Open()` et `Close()` de "`filesystem.cc`" prennent en charge cette table.

À chaque `Open()` on vérifie si le fichier à ouvrir existe, et si tel est le cas, on tente de prendre le verrou d'ouverture et de l'ouvrir. Cet appel est donc potentiellement bloquant. À l'inverse, `Close()` libère le verrou et supprime la référence à ce fichier de la table si personne ne l'attend (on compte le nombre de tentatives de prise du verrou).

RÉSEAU

Nous n'avons pas eu suffisamment de temps pour implémenter la partie réseau en totalité, nous nous sommes donc arrêtés à la mise en place d'un système d'attente active que vous pourrez trouver ci-dessous, et nous avons tenté d'implémenter une technique de découpe des messages que nous n'utilisons pas dans le système final. Toutefois, il était prévu de mettre en place un système similaire à TCP/IP pour la découpe des messages en paquet. À savoir, attribuer un numéro de séquence et d'acquittement afin de reconstituer les messages et contrôler les pertes. Le tout couplé à un système de buffer d'émission afin d'enregistrer les paquets pour réémission en cas de non réception d'un acquittement après TEMPO ticks.

Pour des messages de longueur variable, nous avons tenté d'implémenter le protocole suivant :

Du côté de l'expéditeur :

- Envoie d'un message contenant la taille de ses données et attend un message d'acquittement.
- Une fois l'acquittement reçu, l'expéditeur commence à envoyer les paquets en utilisant `Send()`.
- L'expéditeur attend pour un message d'acquittement.

Du côté du destinataire :

- Le destinataire est informé de la taille du message à recevoir.
- Une fois la taille reçue, il envoie un message d'acquittement et boucle tant que la taille reçue est inférieure à la taille totale.
- À chaque fois qu'une nouvelle partie du message est reçue, il envoie un autre message d'acquittement.

SYNCHRONISATION

À partir de l'étape 6 nous avons implémenté l'intégralité des mécanismes de synchronisation. Les verrous ont été implémentés à l'aide d'un sémaphore initialisée à un et les variables-conditions ont été implémentées à l'aide desdits verrous.

Afin d'implémenter les variables-conditions, nous avons créé une file d'attente contenant des pointeurs de thread, indiquant lesquels sont bloqués sur cette condition. Cette file nous permet de faire facilement appel à la méthode `Broadcast()` et ainsi de signaler à l'intégralité des threads de concourir à nouveau pour l'obtention du verrou.

Nous avons par la suite implémenté un système d'attente limitée. Pour ce faire, nous avons ajouté une nouvelle liste de pointeurs de Thread dans la classe `Scheduler`, ainsi qu'une méthode `temporarilySleep()` dans la classe `Thread` et deux nouveaux attributs : un entier `wakeUpTime` permettant de spécifier un temps de réveil en nombre de ticks, et un booléen `signaled`, permettant de connaître l'origine du réveil de ce thread. Nous avons également activé les interruptions du `Timer` afin d'avoir de pouvoir mesurer le temps.

Une méthode `temporaryWait()` a également été implémentée dans `Condition`. Celle-ci fait en interne appelle à `temporarilySleep()`, qui va inscrire le thread appelant dans la liste des threads endormis de `Scheduler`. Cette fonction fait ensuite appel à `Wait()` qui va atomiquement relâcher le verrou passé en paramètre, et inscrire également le thread appelant à la liste des threads bloqués sur cette condition.

Enfin, une méthode `WakeUpReadyThread()` a été incorporée dans `Scheduler` afin de réveiller tout thread dont le temps d'endormissement est dépassé ou qui a été marqué comme `signaled` par la méthode

Signal()).

Afin de prendre en compte l'écoulement du temps, nous avons modifié `TimerInterruptHandler()` afin qu'il fasse appel à `WakeUpReadThread()` à chaque interruption du Timer. Cela signifie qu'à partir du moment où un thread a dépassé son temps d'endormissement, il sera réveillé à la prochaine interruption d'horloge (et non pas immédiatement).

Il est également important de préciser que pour un meilleur calcul du temps, nous avons jugé plus intéressant de nous baser sur le nombre de ticks système. Cela nous permet en effet de prendre en compte le temps passé en code utilisateur et en code noyau.

Veuillez noter qu'il reste un bug que nous n'avons pas eu le temps de corriger au sein du code. En effet les attentes limitées ne fonctionnent à l'heure actuelle que lorsqu'un seul thread est bloqué sur une même condition. Autrement, plusieurs threads font parties de la liste des threads bloqués dans la variable-condition, et ceux-ci ne sont pas effacés de cette liste lors d'une interruption du timer, amenant des problèmes potentiels.

Organisation

Lors de la première journée, nous avons rapidement discuté de l'organisation. Nous avons ainsi décidé des canaux de communication, de la plateforme de gestion du code, du planning, des responsabilités et de la structuration des journées et des étapes.

Communication

Au niveau de la communication, nous avons décidé d'utiliser à la fois Discord pour la communication du groupe, et d'utiliser Mattermost pour la communication avec l'équipe enseignante.

Séparer la communication sur deux systèmes distincts présente l'avantage de ne pas noyer inutilement les messages d'aide de Mattermost, tandis que nous sommes davantage familier avec Discord.

Hébergement du code

Nous avons ensuite décidé de la plateforme d'hébergement. Notre choix s'est rapidement porté sur Github. D'une part car il s'agit de la plateforme d'hébergement la plus utilisée mais également parce que l'un des membres de notre équipe était particulièrement à l'aise avec la plateforme et plus à même d'organiser le code et de régler les problèmes pouvant survenir.

Workflow

Sur Github, nous avons organisé le travail grâce aux fonctionnalités présentes sur la plateforme. Nous avons commencé par créer un projet, puis un ensemble d'étapes pour jalonner nos progrès et garder une trace du travail à effectuer. Nous avons également créé un ensemble de labels afin de catégoriser plus facilement nos Pull Requests et nos issues, puis nous avons adopté le workflow suivant : à chaque fonctionnalité ou bug est associée une issue. Chaque personne s'attribue ensuite une ou plusieurs issues afin que tous le monde puisse savoir à tout moment qui fait quoi en cas d'absence dans l'équipe. Enfin, à chaque issue ou groupe d'issues résolu, une Pull Request est créée pour implémenter les nouvelles fonctionnalités. Les issues associées sont ensuite fermées.

Au fur et à mesure de l'écriture du code, nous avons documenté nos fonctions et méthodes conformément aux standards de documentation du code C++ que vous pourrez trouver à ce [lien](#).

PLANNING

Concernant le planning, nous avons choisi de nous réunir sur Discord du lundi au samedi entre 10H et 18H afin que tous le monde ait un cadre de travail clair et fixe. Chaque début d'étapes commence par la lecture des documents affiliés, puis d'une séance de division des tâches, lorsque cela est possible, afin d'accélérer la cadence de travail.

En fin de journée, une fois le travail effectué, nous écrivons les logs et/ou le rapport des tâches ac-

complies. L'ensemble des documents relatifs au rapport peuvent d'ailleurs être trouvé dans le dossier `"doc/report"`.

Enfin, afin de pouvoir développer collaborativement le rapport, nous avons choisi de nous servir de LaTeX via la plateforme Overleaf.

Bug connus

Entrées / Sorties

—

Threads utilisateur

—

Processus

—

Mémoire

—

Réseau

—

Synchronisation

- Problème d'effacement des threads dans la liste des threads bloqués sur un objet Condition. Peut occasionner des problèmes si plus d'un thread attend sur la même condition. Lors d'une interruption timer en effet, on efface pas les threads qui sont potentiellement réveillé par l'interruption dans cette liste.