H-NACHOS : LE RAPPORT

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

Project Système - 2021

H-NACHOS : SPÉCIFICITÉS ET FONCTIONNALITÉS

H-NachOS est un système d'exploitation minimaliste. Il reprend jusqu'ici les fonctionnalités de base de la plupart des systèmes d'exploitation.
Nous nous sommes pour l'instant dédié au développement de systèmes d'entrées/sorties basiques ainsi qu'au développement de processus légers exécutables en concurrence.

Manuel de spécification

Nous allons ici détailler les fonctionnalités accessibles dans H-NachOS.

APPELS SYSTÈME

Cette section détaille les appels système. 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 spécifiée dans le constructeur de `SynchConsole`. Si aucun caractère n'est présent sur l'entrée, l'appel est bloquant. Renvoie un char représentant le caractère lu sur l'entrée.

`void PutChar(char c)`
: Affiche le caractère `c` sur la sortie spécifiée dans le constructeur de `SynchConsole`. PutChar ne gère pas les caractères sur deux octets.

`void GetString(char *s,int size)`
: Lit `size` (au maximum 64) octets sur l'entrée spécifiée dans le constructeur de `SynchConsole` qui seront stockés dans le buffer `s` passé en argument. La lecture s'arrête si on rencontre un EOF ou un retour à la ligne. Le caractère `'\0'` est écrit à la fin du buffer.

`void PutString(char *s)`
: Affiche la chaîne de caractère `s` sur la sortie spécifiée dans le constructeur de `SynchConsole`. La chaine `s` ne devrait pas faire plus de 64 octets, ou elle ne sera pas intégralement écrite. L'écriture s'interrompt au premier `'\0'` rencontré.

`int GetInt()`
: Lit un nombre entier compris entre -2^{31} et 2^{31} sur l'entrée spécifiée dans le constructeur de `SynchConsole`. Si l'entier à lire dépasse la borne minimale, il prendra la valeur minimale. Si l'entier à lire dépasse la borne maximale, il prendra la valeur maximale. Retourne un entier, l'entier lu.

`void PutInt(int d)`
: Affiche l'entier signé `d` sur la sortie spécifiée dans le constructeur de `SynchConsole`. La valeur maximale pouvant être écrite est 2147483647. Si l'entier à écrire dépasse 2147483647, on force l'écriture à 2147483647.

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 ne devrait être appelée que 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 sytème ne devrait être utilisé. 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 ne devrait être appelée que par un thread utilisateur.

Processus

`int ForkExec(char *filename)`
: Permet de créer un nouveau processus à partir de l'exécutable de chemin `filename`. Un processus dispose de son propre espace d'adressage et par conséquent s'exécute complètement parallèlement aux autres processus. Retourne un entier, le pid du processus créé ou -1 s'il ne peut être créé.

`void Halt()`
: Interrompt le système immédiatement, sans se préocuper de la terminaison des processus ou threads encore en cours. Cette fonction ne doit être appelée que par un processus.

`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

Pour les appels systèmes concernant le système de fichiers, l'argument `name` peut être un chemin ou un nom de fichier. Dans le premier cas, on essaie de rejoindre le dernier répertoire du chemin, si on n'y arrive pas, on reste dans le répertoire courant. Un repertoire peut contenir jusqu'à dix références (fichiers ou répertoires).

`void Create (char *name)`
: Crée un fichier nommé `name` ou un répertoire si `name` contient un `'/'` en dernier caractère dans le répertoire courant s'il n'est pas plein et que ce fichier 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.

`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.

`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. Le test est une boucle, elle cesse lorsqu'on entre le caractère `'q'`. N'importe quel caractère peut être lu : chiffres, caractère spéciaux et lettres.

– getint

Permet de lire un entier inférieur à 2^{31} 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 chaine 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}.

– getstring

Permet de lire une chaine de caractères de 10 octets sur l'entrée standard. Cette valeur peut être modifiée dans le fichier test. Cette chaine sera renvoyée sur la sortie standard pour vérifier la bonne exécution. Notons que certains caractère spéciaux s'écrivent sur deux octets, comme `'¤'` ou `'ê'`. Si on saisit 10 fois le caractère `'ê'`, on ne l'écrira que 5 fois.

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.

– putint

Permet d'afficher un nombre compris entre -2^{31} et 2^{31}.

– putstring

Permet d'afficher une chaine de caractères quelconque, d'une taille maximale de 64 octets.

Console synchronisée

Permet d'afficher les charactères 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-step? -sc`

THREADS

Threads utilisateur

Les tests de threads utilisateur sont dans les fichiers `userthreadcreate_*.c` et se différencient par ce qu'ils testent (`fewthreads,join,automaticend,manythreads`). Nous les avons effectués avec la commande `./nachos-step3 -x testfile`.

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

– userthreadcreate_fewthreads

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

– userthreadcreate_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'ordonnancement.

– userthreadcreate_automaticend

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.

– userthreadcreate_manythreads

Permet de tester le nombre maximum de threads en concurrence. Ce nombre est codé dans la variable `NB_MAX_THREADS` dans le ficher `system.h`. Ce test tente de créer 100 threads. Lors d'une exécution sérielle, on retrouve bien 10 threads créés à la fin du test. 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 utilistateur peut être créé. Cela dépend du temps d'execution 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.

Nous avons effectué ce test avec `NB_MAX_THREADS = 10`.

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 thread utilisateur, tout en s'assurant de leur bonne cohabitation en mémoire. On vérifie par la même occasion si la synchronisation entre processus est correcte.

– process2threads + process2processes

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*12 = 144 fois l'entier 1. Pour passer ce test il nous aura fallut augmenter la taille de la mémoire physique. Elle doit posséder pour ce test environ 400 pages minimum car nous n'effectuons pas d'allocation dynamique. Nous l'avons fixé à 512. Notons que chaque threads dispose d'une pile de 2 pages. C'est pourquoi il nous faut au minimum 144(threads) * 2(nombre de page pour la pile) = 288 pages uniquement pour les threads. S'ajoute à ca 11 espaces d'addressage 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.

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
- `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

Ces fichiers demandent d'être copiés dans le système de fichier de H-NachOS pour être testés. Il est recommendé de formater le disque avant d'effectuer cette opération.

– filesys_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.

– filesys_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 (ces répertoires doivent être créés au préalable).

Une boucle de taille 50 tente d'ouvrir un fichier parmis 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.

– filesys_lock

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 entammée, 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 relaché 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 étendu 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()`.

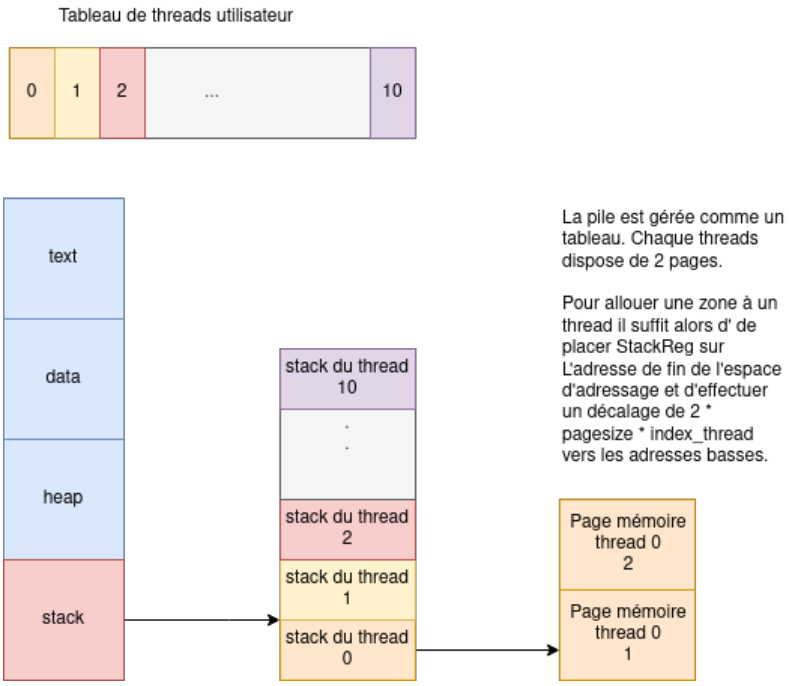

FIGURE 1 – Gestion de la pile

L'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 multipliant simplement l'index de ce thread par $2 \times PageSize$ puis en l'ajoutant à l'adresse de début de la pile du processus. 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 : taille_pile $= 2 \times$ PageSize $\times$ NB_MAX_THREADS. Lorsque le tableau est plein, la création de thread utilisateur supplémentaire et 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()`.

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*.

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 descrites ci-dessus nous vous laissons vous référer à la documentation interne du code trouvable dans `"userprog/addrspace.cc"`.

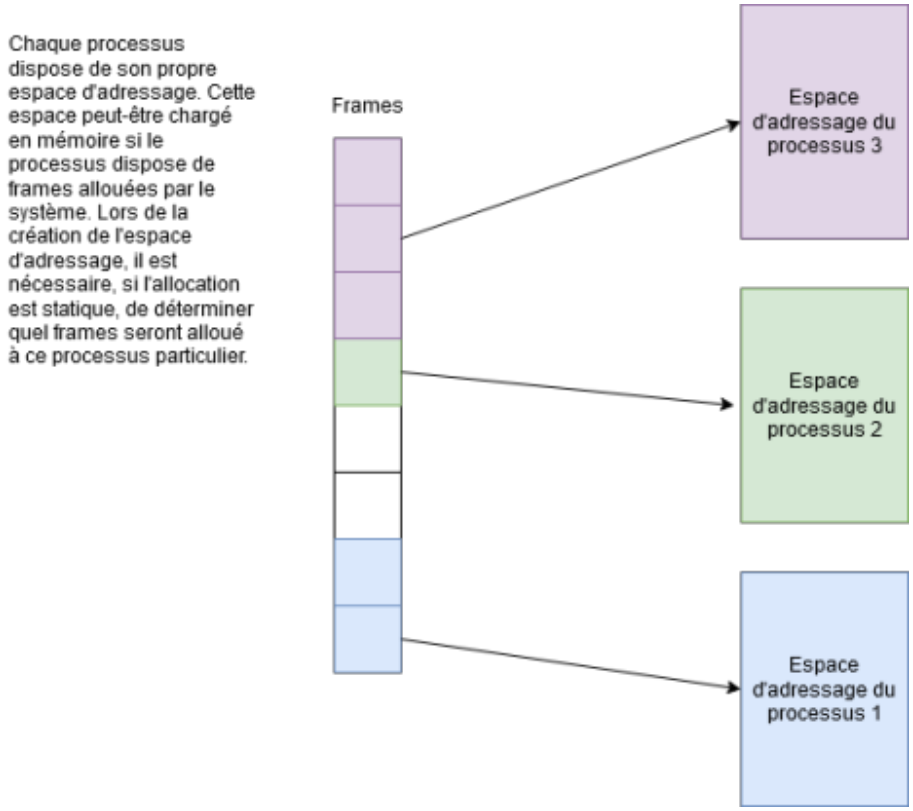

FIGURE 2 – Gestion de la mémoire

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 synchronistation 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 entete 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 `"filesys.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, nous nous sommes donc arrêtés à la mise en place d'un système d'attente active que vous pourrez trouver ci-dessous. Toutefois, il était prévu de mettre en place un système similaire à TCP/IP pour la découpe des messages en paquet. A savoir, attribuer un numéro de séquence et d'acquittement afin de reconstituer les messages et de contrôler la perte. 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 pendant TEMPO ticks.

SYNCHRONISATION

À partir de l'étape 6 nous avons implémenté l'intégralité des mécaniques de synchronisation. Les verrous ont été implémenté à l'aide de 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 concourrir 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 `temporarily-Sleep()` 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 appellant 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 appellant 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 immédiatemment).

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.

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 règler 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 créer un ensemble de label 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'accèlèrer la cadence de travail.

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