IFT320

SYSTÈMES D'EXPLOITATION

Travail pratique 3

Travail en équipe:
Pour les groupes de **deux** étudiants exactement, les parties à réaliser sont :
1-a) b) c) d) e) f)
2-a) b) c)
3-a) b)

Pour les étudiants voulant être **seuls**, les parties à réaliser sont :
1-a) b) c) d) e) f)
2-a) b) c)

Pour les étudiants voulant être **en groupe de trois**, toutes les parties sont à faire. L'omission des parties supplémentaires vous pénalise de 20% de la note.

Les parties supplémentaires peuvent être faites en bonus par les personnes seules ou les équipes de 2. Les bonus sont plus payants pour les personnes seules (note pondérée sur un total plus bas).

Mode de remise: électronique

Les trois derniers travaux pratiques du cours IFT320 portent tous sur NachOS (*Not Another Completely Heuristic OS*), un système d'exploitation minimaliste à but pédagogique. Au cours de ces TPs, vous aurez à modifier différentes parties du système afin de leur rajouter des fonctionnalités.

À travers les années, une certaine quantité d'explications et de documentation a été rassemblée à propos de NachOS. Lisez le document d'introduction à NachOS pour l'ancien cours IFT518 (*ancien_nachos_ift518.pdf*).

Cependant, notez bien que dans ce document **les anciens énoncés des travaux et les dates de remise indiqués ne sont plus valides**. Les dates de remise pour cette session sont celles indiquées dans le plan de cours, et les énoncés des travaux se trouvent ci-dessous.

Les tests seront faits dans l'ordre dans lesquelles les questions sont posées; il est donc fondamental que vous complétiez chaque question avant de commencer la suivante. Pour chaque travail pratique, vous devrez soumettre par **turnin web** vos sources modifiées de NachOS. La structure de répertoires du dossier « nachos » qui vous est fourni **doit demeurer intacte**. Vous devez remettre le répertoire « nachos » au complet avec ses sous-répertoires.

Toute demande de resoumisson ou de recorrection se verra octroyer une pénalité de 10% de la note.

Finalement, dans tous vos travaux, souvenez-vous de **ne jamais modifier le répertoire machine et son contenu** !
Nous copierons notre version à la place de la vôtre lors du train de test pour s'assurer de cela.

Travail Pratique 3 : gestion des processus

N.B. Si vous préférez l'approche directe pour réaliser ce devoir, passez au tutoriel qui se trouve plus loin dans ce document, ou regardez les commentaires au début du fichier /userprog/exception.cc puis revenez lire ce qui suit lorsque vous aurez besoin d'être dirigés.

Processus noyau

Dans son état initial, NachOS est capable d'exécuter des processus, avec quelques restrictions. Tout d'abord, il peut simuler plusieurs processus concurrents, mais ils exécutent tous des parties du code de NachOS même, on peut les qualifier de « processus noyau ». Ces processus sont représentés par la classe Thread (bloc de contrôle de processus tel que vu en classe). Attention! Ici, la classe Thread ne représente pas des fils d'exécution (terme Thread dans la littérature) mais bien des processus. *Chaque utilisation du terme Thread dans ce document réfère à un processus, pas un fil d'exécution.*

Afin de réaliser cette simulation, NachOS est capable de sauvegarder l'état de son exécution (zone de sauvegarde au début de l'objet de type Thread). Pour faire comme si on avait plusieurs processus, on n'a qu'à sauvegarder une copie des registres SPARC (incluant le compteur ordinal) dans chaque objet Thread représentant une exécution concurrente de NachOS. Pour changer le processus en cours, il suffit de sauvegarder les valeurs courantes des registres SPARC dans le Thread courant (currentThread), puis de copier les valeurs des registres sauvegardées dans le Thread qu'on veut exécuter dans les registres SPARC. Finalement, on fait un branchement à l'adresse conservée dans la copie du compteur ordinal sauvegardée dans le Thread qu'on veut exécuter et le tour est joué.
Ces changements sont réalisés et contrôlés par la classe Scheduler (dans scheduler.cc/.h). Une fonction spéciale qui réalise ces opérations en assembleur s'appelle SWITCH, et se trouve implantée en assembleur dans switch.s.

Tous les changements de processus se font dans la méthode Run de la classe Scheduler. La processus courant est toujours le pointeur global currentThread.
Tout code exécuté dans le programme nachos est un processus (currentThread est toujours une variable valide à partir de sa création). Le processus courant change

toujours dans scheduler->Run(). Cependant, beaucoup de situations peuvent déclencher un changement de processus (donc un appel à scheduler->Run).

Processus utilisateurs (applications)

En plus de supporter des processus noyau, la version actuelle de nachos supporte des processus utilisateurs (ou applications).

Ces processus et leurs états sauvegardés sont représentés par la classe AddrSpace (addrspace.cc/.h).

Un processus noyau (Thread) peut se voir associé un processus utilisateur (AddrSpace) si on le désire.

Un processus utilisateur (AddrSpace) ne peut pas s'exécuter sans avoir son processus noyau (Thread) correspondant.

Lors de l'exécution d'un processus utilisateur, la variable currentThread pointe sur l'objet Thread correspondant au processus noyau associé.

Le code des processus utilisateurs est exécuté dans un environnement simulé (machine virtuelle MIPS, machine.cc/.h). Un processus noyau contenant un processus utilisateur va typiquement avoir pour seul objectif d'exécuter le code du processus utilisateur.

Pour ce faire, il crée un objet AddrSpace, l'initialise, place le simulateur dans le bon état et démarre la simulation (machine->run()). Ceci est réalisé dans la fonction **StartProcess**, dans le fichier progtest.cc.

La simulation est mise en pause uniquement lorsque des interruptions surviennent, notamment des erreurs de programme (bus error, etc) ou des appels système (SysCall). Lorsque ceux-ci se produisent, la fonction ExceptionHandler de exception.cc sera appelée et la simulation sera mise en pause, le temps que vous réalisiez le travail nécessaire. Si la situation le permet, la simulation du processus utilisateur reprendra (fin de la fonction ExceptionHandler), sinon, le processus courant perdra le contrôle de l'UCT (Exit, Yield, erreur de programme, Join, etc) et le planificateur (Scheduler) donnera le contrôle à un autre processus.

Présentement, **un seul processus utilisateur** peut occuper la mémoire de la machine virtuelle à la fois. Vous pouvez charger un processus utilisateur à l'aide de la commande nachos –x [nom d'executable].

Un fichier exécutable pour nachos est un programme compilé pour être exécuté sur le simulateur MIPS. Plusieurs de ces fichiers sont fournis dans les répertoires /test et

/test_tp3. Dans /test, seul le fichier « halt » va fonctionner correctement d'ici à la fin de ce devoir. Dans /test_tp3, avec un peu de chance, TOUS les fichiers devraient fonctionner d'ici la fin.

Le code source des exécutables chargeables sur nachos est écrit en C, pas en C++!!! Un compilateur spécial les transforme en exécutables dans un format que nachos peut charger dans son simulateur MIPS.

Le code simulé des processus utilisateurs, ainsi que leurs données se trouvant chargées dans le simulateur MIPS peuvent être vues comme étant dans dans des espaces d'adresse utilisateurs, tandis que le code de nachos sur lequel vous travaillez, qu'exécutent les objets Thread, peut être vu comme étant dans l'espace d'adresse noyau. Le transfert de données entre les deux implique des accès à la mémoire de la machine virtuelle MIPS (ReadMem et WriteMem).

Tâches à réaliser

Beaucoup d'étapes doivent être franchies avant de faire de NachOS un système avec une gestion des processus un peu plus respectable. Le travail se divise en trois parties importantes :

1) appels système pour un seul processus
2) appels système pour plusieurs processus
3) algorithmes de planification

Chaque partie contient plusieurs numéros. Pour chaque numéro, des tests sont fournis, afin de vous permettre de vérifier leur réalisation. Ces tests sont les mêmes qui seront faits sur le travail que vous allez remettre. Les noms mentionnés pour les tests sont des scripts qui exécutent un ou plusieurs processus utilisateurs sur nachos (nachos –x …).

Vous pouvez taper ces commandes directement dans le répertoire /userprog. Si vous voulez savoir quels programmes sont chargés par un test en particulier, vous pouvez afficher le script avec la commande « cat » (cat tjoin par exemple, pour voir la commande faite par le test « tjoin »). Les fichiers de code source des tests sont tous dans le répertoire /test_tp3 et continennent généralement des commentaires qui expliquent ce que les tests devraient faire.

Partie 1 – appels système pour un processus

N.B. : cette partie doit être réalisée au complet par toutes les équipes

Les modificatons pour cette partie sont presque entièrement réalisées dans le fichier exception.cc.

a) Appel Système Exit
Script de test : texit
Réalisation : 20 minutes (y'a un peu de lecture, quand-même…)

Faites un tour dans **ExceptionHandler** (exception.cc) et regardez un peu la structure de conditions et de switch/case. C'est la routine d'interruption pour les appels système et les erreurs de programme, et ce que vous êtes en train de lire, c'est le diagnostic d'interruption.
Les définitions des codes des divers appels système se trouvent dans **syscall.h**, en passant.
L'appel système le plus facile à implémenter est **Exit**, terminer un processus (fin normale). Faites un tour dans thread.cc/.h et trouvez la fonction la plus appropriée pour mettre fin au processus courant. Ça ne ferait pas de tort non plus d'ajouter un printf qui indique ce qui vient de se passer pour voir si votre Exit fonctionne bien. Ça risque d'être utile plus tard également, quand plusieurs processus vont démarrer et se terminer dans un certain ordre.

b) Erreurs de programmes
Script de test : terror
Réalisation : 20 minutes

Les programmes utilisateurs peuvent parfois faire des erreurs et c'est au système d'exploitation de les détecter et de les traiter. Ces erreurs sont générées par la machine virtuelle MIPS, lorsque le code simulé fait des erreurs. Allez faire un tour dans machine.h et regardez l'enum **ExceptionType**, il est assez informatif à ce sujet…

Syscall est un type d'exception, le seul qui ne représente pas une erreur (pour lequel vous avez un paramètre, qui est le sujet du switch-case).

Que devrait faire le système d'exploitation en cas d'erreur?

SyscallException : traité par le switch-case; diagnostiquer et traiter l'appel système en question. Sujet des questions suivantes.

PageFaultException: sujet du tp4. Pour l'instant, si ça arrive, blue screen of death (nachos au complet doit crasher et vous devez vous en rendre compte au plus vite). Ne va vous arriver que si VOUS avez fait des erreurs substancielles plus loin dans votre implémentation. Vous avez pris une hache au lieu d'un scalpel pour travailler dans AddrSpace. Corruption mémoire. Nasty stuff.

ReadOnlyException: Accès mémoire illégal. Le processus courant ne peut pas continuer.

BusErrorException: Accès mémoire illégal. Le processus courant ne peut pas

continuer.

AddressErrorException: Accès mémoire illégal. Le processus courant ne peut pas continuer.

OverflowException: Dépassement de la valeur entière maximum. C'est pas grave.

IllegalInstrException: Instruction indécodable ou réservée. Le processus courant ne peut pas continuer.

NumExceptionTypes: Vraiment? C'est juste le nombre de valeurs dans l'enum. Ça a pas rapport.

NoException : Aussi utile qu'un Branch Never. Si ça arrive, blue screen of death (nachos au complet doit crasher et vous devez vous en rendre compte au plus vite). Ne va vous arriver que si VOUS avez fait des erreurs substancielles plus loin dans votre implémentation. Vous avez probablement bousillé l'état interne de la machine MIPS simulée (corruption mémoire). Probablement le résultat d'un duel au sabre laser dans AddrSpace.

c) CopyFromUser et Write (journal noyau)

Script de test : tlog
Réalisation : 30 minutes à 2h

Les processus utilisateurs ne peuvent pas faire grand-chose d'utile présentement, à part éteindre le système (halt), terminer (exit) ou planter (erreurs).

Afin de pouvoir communiquer avec le monde extérieur, il faut leur offrir l'appel système Write. Cet appel écrirait normalement dans un fichier, mais il faudrait implanter le support pour quelques autres appels (read, open, create) du même coup, ce qui est une trop grosse bouchée.

On va donc implanter un journal noyau (log) à l'aide de l'appel système Write. Lorsqu'un processus utilisateur demande un Write, si son OpenFileId (poignée de fichier) est **ConsoleOutput**, alors ça veut dire qu'il désire afficher de l'information dans le journal noyau.
C'est quoi le journal noyau? Ben, c'est juste un printf. En fait, non, c'est un appel à DEBUG avec le paramètre 'l' (qui est un printf activé seulement si nachos est exécuté avec l'option **–d l**).

Ça paraît simple : on demande un Write sur ConsoleOutput, il suffit de faire un énoncé DEBUG, mais il y a un problème : les données à afficher sont dans l'espace d'adresses du processus utilisateur.

Pour les récupérer, il faut implanter CopyFromUser, une routine qui sert à récupérer

des données dans l'espace d'adresses utilisateur et les ramener dans une variable noyau (un tampon de la bonne taille).

Tout d'abord, il faut récupérer les paramètres de l'appel système. Ceux-ci sont dans les registres r4, r5 et r6 respectivement (utilisez **ReadRegister**). Le paramètre qui représente les données à afficher est une adresse où celles-ci commencent dans le processus utilisateur, mais c'est une adresse valide seulement dans le contexte du MIPS simulé.

La traduction d'une telle adresse se fait relativement simplement, il suffit de demander à la machine de lire la donnée à cette adresse (**ReadMem**). Attention, lisez bien la description de cette fonction (machine.cc/.h). Vous ne pouvez pas lire la mémoire du MIPS comme bon vous semble, il y a des règles à respecter.

Les données ne seront pas tout à fait transférées de la forme qui vous plaît, un petit retour sur vos connaissances de la représentation des types de données et des règles de conversion de type (sans oublier le type Word32 déclaré dans exception.cc) devrait permettre de trouver une solution.

Finalement, il est important de constater que c'est le premier appel système que vous implantez qui retourne le contrôle au processus utilisateur après sa réalisation. Pour qu'un appel qui se termine correctement puisse être pleinement réalisé, il faut manuellement avancer le compteur ordinal du MIPS simulé (fonction **incrementPC**). Ceci est valide pour tous les cas d'appel système pour lesquels le processus utilisateur va un jour continuer son exécution (peut importe si c'est remis à plus tard à cause d'un changement du processus courant via scheduler).

IMPORTANT : La réalisation de cette partie est cruciale à la réussite de la majorité des tests suivants. Les tests les plus avancés (plusieurs processus et algorithmes de planification) utilisent presque tous le journal noyau pour afficher des résultats. Assurez-vous d'avoir réalisé ce numéro proprement. N'ajoutez pas d'affichages inutiles dans ce numéro. Seules les données que le programme utilisateur souhaite afficher devraient apparaître lors d'un Write dans le journal noyau. Pas de décoration SVP.

Pour tous les numéros, si vous souhaitez mettre des affichages de traces pour fins de débuggage, rappelez-vous de les retirer avant de soumettre votre travail. Plusieurs des tests avancés chargent des programmes qui font des boucles infinies, si vous avez laissé des traces, votre devoir va polluer les résultats du script de correction et il devra être corrigé manuellement, ***ce qui vous octroiera une pénalité de 10%.***

d) Create
Script de test : tcreate
Réalisation : 20 à 30 minutes

Il est temps d'implanter les services pour la gestion des fichiers. Notez bien, le système

de fichiers tel qu'implanté lors du TP2 ne rentre pas en ligne de compte pour ce devoir. La version du système de fichiers qui sera utilisée se trouve complètement implantée (corps de méthode et tout!) dans le début du fichier filesys.h. Les fichiers que vous allez créer, ouvrir, lire et modifier seront tous directement dans votre répertoire unix sur tarin (chemins relatifs au répertoire où s'exécute nachos).

Aucune gestion de la table des fichiers ouverts ne doit être implantée… ce qui implique une petite perte de mémoire qu'on devra tolérer (objets OpenFile qui seront créés, mais jamais détruits dans le cas de l'appel système Open).

Tout d'abord, l'appel système Create vous envoie en paramètre une chaîne de caratères dont la taille est inconnue. Ceci implique que votre fonction CopyFromUser actuelle, qui a besoin de savoir la taille, ne peut pas fonctionner. Implantez une seconde version qui est spécifique à la copie de chaînes de caractères. Le caractère de fin de chaîne a la valeur 0.

Ensuite, il suffit de faire appel aux fonctionnalités du système de fichiers nachos (variable fileSystem) pour créer le fichier.

e) Open et Write (fichier)
Script de test : twrite
Réalisation : 20 à 30 minutes

Afin de réaliser quelque-chose d'intéressant, on doit pouvoir écrire dans les fichiers qu'on a créés. Pour ce faire, il faut d'abord permettre l'ouverture des fichiers, puis ensuite, l'écriture dans autre chose que le journal noyau.

Écrivez d'abord le code pour l'appel sytème Open. Celui-ci devrait produire un objet de type OpenFile et retourner la valeur de ce pointeur à l'utilisateur (sous forme de OpenFileId).

Ensuite, modifiez votre appel système Write pour supporter l'écriture lorsque vous recevez un OpenFileId autre que ConsoleOutput (qui servait à envoyer des messages au journal).

f) CopyToUser et Read (fichier)
Scripts de test : tread et treadwrite
Réalisation : 30 minutes à 1h30

Finalement, on arrive à un point où on aura réalisé l'implantation de toutes les primitives permettant de faire l'ultime programme utilisateur par excellence, copier le contenu d'un fichier vers un autre.

La lecture se fait comme suit :

1-Récupération des paramètres du Read;

2-Lecture des données à partir du fichier demandé (paramètre dans r6) et transfert dans un tampon noyau;
3-Transfert des données du tampon noyau au tampon utilisateur (paramètre dans r4).

L'étape 3 requiert la fonction CopyToUser (contraire de CopyFromUser). Son implantation passera nécessairement par la fonction WriteMem de la machine virtuelle (contraire de ReadMem).

Partie 2 – appels système pour plusieurs processus

N.B. : les parties a) b) et c) doivent être réalisées par toutes les équipes. Les équipes de 3 doivent également faire les parties d) et e). Celles-ci sont des bonus (5% chacune) pour les autres.

Les modifications pour cette partie sont surtout dans exception.cc, et dans addrspace.cc, avec quelques-unes dans thread.cc/h.

a) Appel Système Exec

Script de test : texec
Réalisation : 30 minutes à 1h.

Afin de permettre l'exécution plus fluide de programmes utilisateurs et d'éventuellement partager le temps sur l'UCT entre ceux-ci, il faut permettre à des processus utilisateurs de demander l'exécution d'autres processus utilisateurs.

Ce service est implémenté par l'appel système Exec. Pour démarrer un nouveau processus, on appelle Exec avec le nom du fichier exécutable en question (doit être un exécutable en format NachOS, comme ceux dans /test et /tp3_test). Une priorité doit également être fournie pour calibrer les algorithmes de planification (sera utile plus tard, vous pouvez l'ignorer pour l'instant).

Présentement, la planification est faite avec l'algorithne FCFS, premier arrivé, premier servi. De plus, il n'y a pas de situation qui va mettre un processus dans l'état bloqué!
Ceci implique qu'une fois qu'il prend le contrôle de l'UCT, un processus s'exécute au complet avant de laisser la place à un autre. Par exemple, si le programme P1 commence à s'exécuter et fait Exec(P2), le contrôle de l'UCT n'est pas passé à P2 immédiatement, celui-ci est placé dans la file des processus prêts et sera exécuté quand P1 aura terminé.

Que doit faire votre appel système exec?

1-Récupérer les paramètres (r4 : nom du programme à charger, r5, priorité).
2-Créer un processus noyau (Thread) qui sera chargé d'exécuter le programme en question sous forme de processus utilisateur (AddrSpace).
3-Initialiser le processus noyau pour que lorsqu'on lui donne le contrôle de l'UCT, il charge et exécute le programme demandé sur le simulateur MIPS.

4-Placer le nouveau processus noyau dans la file des processus prêts.

Bien que tout ceci paraîsse compliqué, vous avez deux fonctions relativement magiques qui font déjà la majorité du travail : **Thread ::Fork** et **StartProcess (dans fstest.cc).**

Ces deux fonctions n'ont pas besoin d'être modifiées, seulement appelées correctement... et c'est là que ça se corse.

L'objectif de Thread ::Fork (que vous allez constater si vous lisez le code) est de donner une adresse initiale à la copie de compteur ordinal du Thread, pour qu'on sache quoi exécuter lorsque le Scheduler le choisit pour l'exécution. Fork prend donc une adresse dans le code de nachos (pointeur de fonction) en paramètre. Si cette fonction reçue en paramètre doit elle-même recevoir un paramètre, alors Fork lui transmet sous forme d'un int (devrait se transtyper sans problème si votre paramètre est quelque-chose qui rentre dans un mot de 32 bits... int ,float, pointeur, etc).

Une fois les préparations faites, Fork place le processus dans la file des processus prêts (scheduler->ReadyToRun). Éventuellement, ce processus sera choisi pour l'exécution et la routine SWITCH va copier ses valeurs initiales de registres dans les registrses SPARC et brancher sur sa copie du compteur ordinal, qui contient l'adresse de la fonction en question.

Que devrait faire un nouveau processus noyau qui a pour objectif d'exécuter un processus utilisateur lorsqu'il démarre? Charger l'exécuable contenant le programme dans la mémoire du MIPS et démarrer la simulation. Heureusement, toute la procédure est déjà réalisée dans StartProcess...

ATTENTION!
À cette étape, vous n'avez pas encore modifié le chargement de programmes dans la machine MIPS simulée. Un programme utilisateur ne se voit chargé que lorsque son processus noyau associé (Thread) prend le contrôle de l'UCT. Avec l'algorithme FCFS et aucune condition bloquante, ça arrivera uniquement après que le programme précédent ait complètement terminé. Ceci implique que le nouveau programme utilisateur peut carrément écraser l'ancien dans la mémoire du MIPS simulé sans aucune conséquence grave. Vous allez devoir améliorer ça dans les numéros suivants.

b) Appel système Yield et chargement de plusieurs programmes en même temps
Script de test : tspace et tyield
Réalisation : 2h à 32767h.

Cette partie du devoir risque d'être la plus problématique. Si vous la réalisez mal, vous

allez rencontrer des problèmes un peu partout (peut briser les numéros précédents, ou causer des erreurs qui ne seront révélées que dans les numéros suivants).

Les modifications que vous allez faire pour cette partie sont majoritairement dans addrspace.cc, plus précisément, dans le constructeur d'AddrSpace. Allez y jeter un coup d'oeil et revenez lire la suite.

Pour réalisez cette partie, il faut comprendre comment la mémoire du MIPS simulé est représentée.

La mémoire du simulateur est en fait un grand tableau d'octets, conservé dans l'objet **machine** (machine->mainMemory). Lire et écrire dedans paraît assez simple vu que la la variable en question est publique, mais vous ne devriez JAMAIS faire ça.

Bon, vous allez me dire : oui mais c'est précisément ce qui est fait dans AddrSpace ::AddrSpace(). Oui, mais c'est précisément la raison pour laquelle on n'arrive pas à charger deux programmes en même temps dans la mémoire du MIPS.

La mémoire du MIPS est divisée en blocs, qu'on appelle des « pages physiques ». Le terme technique qu'on utilise habituellement pour ça est « cadre ». En anglais, vous verrez « physical page » ou « frame ».

Un programme utilisateur est également divisé en blocs, qu'on appelle « pages virtuelles ». Le terme technique qu'on utilise habituellement pour ça est tout simplement « page ». En anglais, vous verrez « virtual page » ou « page ».

Chaque morceau du programme qu'on veut exécuter (page virtuelle) doit être chargé dans un morceau de la mémoire du MIPS (page physique). Tous les morceaux, virtuels ou physiques, ont toujours tous exactement la même taille.

Une page physique peut être assigné à un processus comme bon nous semble. Pour réaliser cette association, on a besoin d'une **table de pages** (PageTable). C'est un tableau associatif qui dit quelle page physique contient les données de chacune des pages virtuelles du programme (donc, où sont chargés les morceaux du programme).

Chaque processus utilisateur a sa propre table de pages (contenue dans l'objet AddrSpace qui représente votre processus utilisateur).

Présentement, le constructeur d'AddrSpace crée une table de pages pour votre processus utilisateur, mais assume que le processus sera chargé exactement au début de la mémoire physique, et que les morceaux seront toujours dans l'ordre, donc, que les pages virtuelles 0 à N seront chargées dans les pages physiques 0 à N. Le problème, c'est que si un premier processus a déjà occuppé les pages physiques 0 à N, on va écrire par-dessus en chargeant notre nouveau processus.

Il faut donc pouvoir se rappeler de quelles pages physiques sont déjà utilisées.

Pour ce faire, vous avez besoin d'une variable qui sera globale à tous les AddrSpace (variable de classe, ou juste une variable globale dans addrspace.cc), vous permettant de déterminer quelles pages physiques sont libres et peuvent être assignées à votre nouveau processus.

Comme vous allez charger chaque processus au complet lors de la construction de votre chaque AddrSpace, ceux-ci seront toujours dans un ensemble de pages physiques contigu (genre, P1 occupe 0 à 11, P2 occuppe 12 à 21, P3 occuppe 22 à 30, etc). Une simple variable indiquant où est la prochaine page physique libre peut faire l'affaire... si vous ne vous attendez pas à réassigner les pages qui sont libérées lorsqu'un processus termine!!

ATTENTION!!!
Juste après l'initialisation de la table de pages, vous devez appeler la fonction **RestoreState**. La suite de votre constructeur risque de demander à la machine d'utiliser la table de pages fraîchement initialisée, il faut donc remplacer le pointeur de table de pages anciennement utilisé par celui de votre nouvelle table. Si vous avez réalisé votre appel système Exec tel que prescrit, alors vous ne devriez pas avoir de problème. Vous ne devriez pas faire ça si le currentThread n'est pas le Thread qui possèdera l'objet AddrSpace en cours d'initialisation.

La majorité des tests ne requièrent pas la réutilisation de mémoire préalablement occuppée. Cependant, des parties bonus ou pour les équipes de 3 en ont besoin. si vous voulez directement faire la bonne implantation, lisez la section concernant le remplissage de la mémoire (numéro 2 d)) avant de procéder.

Bon, vous avez réussi à associer des pages physiques libres à votre nouveau processus via sa table de pages... mais vous ne l'avez pas encore chargé en mémoire!

Cette partie du travail est délicate et peut mener à des erreurs catastrophiques.

Heureusement, vous avez plusieurs outils à votre disposition pour éviter de faire des erreurs.
-Une table de pages correctement initialisée (utilisez donc PrintPageTable pour vérifier...)
-Une fonction CopyToUser
-Une implémentation de l'appel système Read

Eh oui, faire le chargement d'un programme, c'est comme si en commençant, le programme vous demandait de faire une lecture de fichier (son exécutable) et de placer dans son espace d'adresses son code et ses données.

Reste à comprendre ce que vous devez lire dans le fichier exécutable, et où ça doit être transféré dans l'espace d'adresses de l'utilisateur.

Eh bien, vous devez interpréter les lignes de code à la fin du constructeur d'AddrSpace pour cela... grosso modo, noffH (l'en-tête de l'exécutable) indique où commencent les sections de données et de code dans le fichier exécutable (inFileAddr), quelle taille ils ont (size) et à quel endroit dans l'espace d'adresses de ils doivent être chargés (virtualAddr).

Si vous aviez une fonction SysCallRead qui pouvait lire où vous désirez dans votre fichier exécutable, le tour serait presque joué!

N.B. Si vous désirez utiliser des fonctions qui sont définies dans exception.cc alors que vous êtes dans addrspace.cc, il suffit de les redéclarer (pas les redéfinir!!!) dans addrspace.cc, en précédant la déclaration du mot-clé **extern**.

ATTENTION! Si la section de données est vide, alors noffH.initData.inFileAddr sera une valeur incorrecte (crash lors de la lecture du fichier). Ça pourrait faire rétroactivement planter certains tests qui fonctionnaient, si vous ne tenez pas compte de ce cas spécial.

Finalement, pour arriver à faire des tests qui vont charger plusieurs programmes à la fois, il doit être possible pour un processus de lâcher l'UCT avant d'avoir terminé. Comme les demandes d'entrée/sortie ne bloquent pas les processus et que l'algorithme est FCFS, on a besoin de tricher et de permettre à un processus de lâcher volontairement l'UCT.

Ce service est l'appel système **Yield**. Il n'est pas particulièrement difficile à implanter, étant donné que cette fonctionnalité est déjà prévue du côté de Thread...

Une fois Yield implanté, vous pouvez lancer tspace et tyield, qui devraient mettre à l'épreuve votre impantation du partage de la mémoire entre les processus utilisateurs.

c) Appel système Join (version incomplète)
Script de test : tminijoin
Réalisation : 30 minutes à 1h.

Afin d'arriver à synchroniser des processus, on a besoin de pouvoir attendre après un processus. Un processus P1 ayant démarré un processus P2 peut décider d'attendre la fin du processus P2 en demandant un **Join** sur celui-ci. Que fait l'appel système Join? Attend la fin du processus reçu en paramètre (son Exit) et récupère le code de sortie (paramètre du Exit).

Pour réaliser cet appel système, vous devez pouvoir manipuler des identificateurs de processus. Un type a été prévu pour ça dans syscall.h : **SpaceId**. Qu'est-ce qu'un SpaceId? Un int, apparemment, mais que devrait-il contenir?

Eh bien, quelque-chose qui identifie un processus de façon unique. C'est comme une

poignée de fichier pour la table des fichiers ouverts et les opérations sur les fichiers, juste une clé qu'on donne à l'utilisateur lors de l'ouverture pour savoir de quel fichier il parle lorsqu'il fait des demandes comme read et write ultérieurement.

Dans le cas d'un processus, l'utilisateur obtient le SpaceId comme résultat de l'appel système Exec, puis utilise cette valeur comme paramètre pour l'appel système Join plus tard.

La meilleure façon (simple, mais non-sécuritaire) d'identifier de façon unique un processus est l'adresse de son objet Thread. C'est unique, et ça a l'avantage de nous éviter d'avoir à implanter une recherche avec l'identificateur de Thread ou une table de correspondance. Suffit de transtyper le SpaceId en Thead* et vice versa. Cependant, gare à vous si la valeur d'un SpaceId est incorrecte! Vous allez passer Go sans réclamer 200$ et aller directement à Segmentation Fault.

Maintenant qu'on sait identifier le un processus, il faut arriver à attendre après un processus. Pour ce numéro, vous pouvez assumer que seuls deux processus s'exécuteront, et une seule demande de Join sera faite.

Un Join typique se déroulerait ainsi :

P1 : Exec P2
P1 : fait des trucs
P1 : Join P2
    -P1 lâche l'UCT
P2 : fait des trucs
P2 : Exit
P2 : lors de son exit, signale P1 qu'il a terminé.

Donc, il faut modifier Exit pour signaler à un processus qui nous attendait qu'on a terminé, et lui envoyer notre code de sortie (paramètre du exit).
Pour seulement deux processus, dans un test très simple, on peut utiliser des variables globales pour se synchroniser. Une qui indique si le processus attendu a terminé (FIN), puis une qui contient le code de retour du processus attendu (CODE).

En pseudo-code, ça donnerait, pour Join :
Si (FIN == faux), PARENT =currentThread, currentThread->Yield()
Si (FIN == vrai) retourne CODE

Pour Exit :
FIN = vrai
CODE = parametre du Exit

ATTENTION! Cet algorithme est incomplet, mais devrait être suffisant pour que tminijoin s'exécute correctement. L'implantation complète du join est le numéro 2 e).

d) Utilisation responsable de la mémoire du MIPS
Script de test : tfull
Réalisation : 1h à 24h.
Cette partie est un bonus pour les équipes de 1 et 2 (5% de la note), mais obligatoire pour les équipes de 3.

Dans le numéro 2 b), on n'a pas prévu qu'un processus pourrait libérer de la mémoire qui serait ensuite réutilisée par un autre processus.

Pour y arriver, il faut avoir une meilleure gestion des pages physiques libres.
Allez faire un tour dans bitmap.cc/h et vous y trouverez la structure de données toute indiquée pour réaliser ce travail.

Cette partie doit être correctement réalisée pour faire l'algorithme HRN (partie 3 c)).

e) Implémentation complète de Join
Script de test : tjoin
Réalisation : 1h à 4h.
Cette partie est un bonus pour les équipes de 1 et 2 (5% de la note), mais obligatoire pour les équipes de 3.

Dans le numéro 2 c), vous avez implanté un Join partiel.
Il s'agit maintenant de réfléchir à comment on peut implanter ce service correctement, si plusieurs demandes de Join sont concurrentes.

Voici les trois situations de Join à 2 processus possibles (si P1 est celui qui exec P2):

1-P1 Join P2 qui n'a pas encore terminé
2-P1 Join P2 qui a déjà terminé
3-P1 ne Join jamais P2
Implantez une structure de données (JoinTable) qui permet de conserver l'information sur les processus qu'un processus a démarrés et leurs codes de sortie. Ces informations seront mises à jour lors d'un Exec et un Exit, et seront récupérées lors d'un Join.

Si vous avez impanté ce genre de structure de données dans le TP2 pour votre table de fichiers ouverts, vous ne devriez pas être trop dépaysé(e).

De plus, un processus qui demande un Join est maintenant bloqué, c'est-à-dire, il ne peut pas être dans la file des processus prêts tant que l'événement qu'il attend (fin d'un processus) n'est pas survenu. Dans la version simple, on utilisait Yield pour retourner à la fin de la file des processus prêts après avoir vérifié si l'événement était survenu. Maintenant, lorsque l'événement survient, on va remettre le processus qui attendait dans la file, mais pas avant.

Partie 3 – algorithmes de planification

N.B. : Les parties a) et b) doivent être faites par les équipes de 2 et 3. La partie c) doit être faite par les équipes de 3. Les autres peuvent la faire en bonus (10% de la note). Si vous faites le travail seul(e), vous êtes dispensé(e) de faire la partie 3 au complet.

Les modifications pour cette partie sont surtout dans scheduler.cc, avec quelques-unes dans system.cc et thread.h/.cc; peut-être aussi dans list.cc/.h. N'oubliez pas que vous pouvez faire afficher le contenu de la file des processus prêts à l'aide de la fonction Scheduler ::Print().

a) Implantation de l'algorithme à priorité statique

Script de test : tprio
Réalisation : 1h à 2h.

Notre gestion des processus est enfin suffisante pour qu'on puisse implanter des algorithmes de planification.

Le plus facile à faire est un algorithme à priorité statique sans réquisition. Grosso modo, lorsqu'un processus est chargé, il est mis en file dans un ordre déterminé par la priorité qu'on lui a donné lors de son Exec. Ainsi, on exécute les processus dans l'ordre de leur priorité.

La priorité est inversement proportionnelle à la valeur du paramètre qu'on donne lors d'un Exec (plus c'est bas, plus c'est prioritaire). La priorité par défaut est 0 (pour le premier processus, qui n'est pas démarré avec un Exec).
Pour pouvoir tester correctement votre algorithme, vous devez modifier system.cc et son traitement des paramètres à l'exécution de nachos. Ajoutez un cas de traitement de paramètres pour qu'on puisse ajouter l'option –a (choisir un algrithme). L'option –a prend un paramètre (0, 1 2 ou 3) qui détermine l'algorithme choisi.

Par exemple :

nachos –x ../test_tp3/ExecTest //Algo par défaut, FCFS.

nachos –a 0 –x ../test_tp3/ExecTest //Algrithme 0, FCFS.

nachos –a 1 –x ../test_tp3/ExecTest //Algrithme 1, Round Robin.

nachos –a 2 –x ../test_tp3/ExecTest //Algrithme 2, Priorité statique.

nachos –a 3 –x ../test_tp3/ExecTest //Algrithme 3, HRN.

Pour savoir comment récupérer les paramètres, regardez le code déjà fait pour les autres options (-x est un bon exemple, il se trouve dans main.cc).

ATTENTION!! Le comportement de votre planificateur (scheduler) va nécessairement être différent selon l'algorithme choisi. Vous devez conserver l'ancien fonctionnement (FCFS) intact pour que tous les anciens tests fonctionnent comme avant. FCFS doit également être choisi lorsqu'on ne spécifie pas l'algorithme (pas d'option –a).

Pour réaliser correctement la priorité statique, c'est une bonne idée d'aller faire un tour dans list.cc/.h pour comprendre comment utiliser la file des processus prêts.

b) Implantation de l'algorithme Round Robin
Script de test : trobin
Réalisation : 1h à 2h.

Il est temps de se mettre le nez dans les réquisitions et les tranches de temps.
Vous devez implanter l'algorithme du tourniquet (Round Robin), qui octroie une tranche de temps fixe à chaque processus et leur réquisitionne l'UCT si celle-ci s'épuise.

Pour calculer le temps, vous allez avoir besoin d'une routine d'interruption et d'une horloge. L'horloge est implantée du côté de machine (timer.cc/.h). Elle appelle une routine d'interruption à toutes les fois qu'un certain nombre d'instructions sur la machine virtuelle MIPS ont été simulées. Elle peut générer des interruptions au hasard, mais vous ne voulez pas que ça se produise (doRandom devrait être FALSE lors de la création du Timer).

Il y a déjà une routine d'interruption pour l'horloge, dans system.cc (TimerInterruptHandler). Il est possible que vous deviez la modifier, ainsi que la création du Timer. **Vous ne pouvez pas modifier la classe Timer, comme touts les autres fichiers du dossier machine**.

L'algorithme Round Robin ne tient pas compte de la priorité. L'algorithme à priorité statique ne tient pas compte du temps qui s'écoule (donc pas de réquisition).

c) Implantation de l'algorithme HRN
Script de test : thrn
Réalisation : 3h à 4h.
Cette partie est un bonus pour les équipes de 1 et 2 (10% de la note), mais obligatoire pour les équipes de 3.

Finalement, c'est l'heure d'implanter un algorithme plus complexe, qui tient compte des priorités et de l'écoulement du temps. L'algorithme tout indiqué pour ça est une variante de l'algorithme HRN vu en classe.

Voici le comportement qui est attendu.

La priorité d'un processus à tout moment est la suivante :

(Priorité statique originale) * (temps de service / temps d'attente).

Le ratio est donc l'inverse de ce qu'on a vu en classe (parce que les valeurs de priorité les plus basses sont les plus prioritaires).
Les valeurs initiales du temps de service et d'attente sont 1.

La priorité de tous les processus doit être recalculée à toutes les interruptions d'horloge. On mettra donc le temps de service du processus en cours d'exécution à jour, puis le temps d'attente de tous les processus qui sont dans la file des processus prêts à jour. Chaque interruption d'horloge compte pour une unité de temps.

Si, en raison de cette mise à jour, un processus a maintenant une priorité plus élevée que celui qui est en cours d'exécution, on doit lui réquisitionner l'UCT.

Pour réaliser correctement cette partie, il est probable que vous deviez ajouter des fonctionnalités à la classe List (attention de ne pas la briser pour les anciens tests et le reste de nachos!!) ou changer la file des processus prêts pour une structure de données que vous jugez plus appropriée.

Mini-Tutoriel

```
cd nachos
cd code
make
```

La commande « make » dans le répertoire « code » compile plusieurs versions de nachos. Il n'est pas nécessaire de toutes les compiler, seule la version présente dans « userprog » nous intéresse pour ce devoir. Si les autres versions ne compilent plus après vos modifications, ce n'est pas grave. Compilez donc dans le répertoire filesys seulement après vos modifications. Il se peut qu'une erreur de compilation survienne dans une des versions de nachos qui ne nous intéresse pas. Ce n'est pas grave. La version de nachos qui nous intéresse est dans /userprog. À l'avenir, compilez uniquement dans /userprog.

```
cd userprog
```

Ici, vous devriez voir tous les fichiers source, les fichiers intermédiaires, et finalement, l'exécutable « nachos ». Cet exécutable est le système d'exploitaton simulé. Chaque fois que vous entrez la commande « nachos », c'est comme si vous démarriez un ordinateur avec le système Nachos dessus. Quand le programme se termine, c'est l'équivalent de dire que l'ordinateur sur lequel Nachos s'exécute s'est éteint.

```
mtest
```

Cette commande compile tous les tests se trouvant dans /test_tp3. Vous ne devriez pas avoir besoin de la refaire si vous ne modifiez pas les tests.

```
nachos –x ../test/halt
```

Ceci exécute le seul test fonctionnel actuellement, c'est un programme qui demande l'arrêt du système.

Finalement, réalisez une première modification dans exception.cc : ajoutez un énoncé case dans le switch de la fonction ExceptionHandler, comme suit :

```
case SC_Exit : printf("EXIT!!!!!!\n");
                    break;
```

Ensuite, recompilez nachos:

```
make
```

Puis, lancez le test pour l'appel système Exit :

```
texit
```

Vous devriez voir apparaître votre message parmi les affichages que fait nachos. Vous devriez ensuite retourner à la description des tâches et réaliser l'appel système Exit correctement (numéro 1 a)).

Bon travail!