CacatOS - Projet de Systèmes et Réseaux

Marc Ducret, Martin Ruffel

May 15, 2017

1 Démarrage de l'OS

On a tout d'abord implémenté le micronoyau en C. Puis nous avons tout d'abord essayé de réussir à booter sur QEMU, que l'on utilisera par la suite, ainsi que BOCHS, utilisé un peu pour débugger certaines situations délicates.

On utilise GRUB2 pour lancer l'OS, lorsque l'on récupère la main en assembleur, on est ainsi déjà en Protected Mode. On se lance alors dans le grand paramétrage initial commandé depuis la fonction `kmain` dans le fichier `main.c`:

- `terminal_initialize` : Initialise l'écran 80x25 que l'on utilisera tout au long du projet. Pour y afficher du texte, nous avons recodé une fonction kprintf, qui par défaut affiche à l'écran.
- `init_gdt` : Définit les segments et leurs autorisations. Puisque nous utilisons le paging, la segmentation n'est pas utile, on ne fait donc que redéfinir des segments user et kernel de 4 Go. Cela ne sert qu'à savoir où est située la Global Descriptor Table résultante, pour ne pas l'écraser par erreur.
- `init_idt` : Cette fonction initialise l'Interrupt Descriptor Table, qui contient les adresses des Interrupt Handlers, appelés lors des diverses interruptions. On redirige ainsi les principales interruptions (ISR 0-32) vers un handler d'erreur, et les IRQ (0-15) vers un autre qui traitera par exemple le clavier, le timer, ... Enfin, on définit une Trap Gate (ce qui permet d'obtenir sur la pile des informations sur le processus interrompu) qui servira pour les appels systèmes, on a choisi l'interruption `0x80`.
- `init_pic` : On envoie quelques paramètres de configuration au Programmable Interrupt Controller, afin d'être sûr que les IRQ sont mappées vers les bonnes entrées de l'IDT (typiquement de 32 à 47).
- `init_timer` : Paramètre l'interruption timer (IRQ 0) pour qu'elle arrive avec la fréquence choisie, ici 1000Hz. Cela initialise aussi le temps et la date, en interrogeant les registres de la Real Time Clock.
- `init_paging` : On arrive dans les choses délicates. Cela initialise le paging, c'est à dire passe d'un addressage physique à un addressage virtuel, chose nécessaire si l'on veut que chaque processus ait l'impression d'être seul dans la mémoire. On crée donc un page directory qui sera celui du premier processus. Dans celui-ci, on map toutes les adresses virtuelles en dessous de la fin du tas du noyau vers elles mêmes (Identity Mapping). Le

noyau sera mappé de même dans tous les page directories créés. On passe ensuite dans ce page directory, ce qui en théorie ne change rien (puisque l'addressage dans la zone kernel n'a pas changé) mais en pratique amène l'interruption Page Fault et une foule de bugs ou reboots.

- `init_disk` : Détecte le disque, l'identifie, et vérifie qu'il est bien de type ATA.
- `init_fs` : Détecte la partition qui est sur le disque, charge ses paramètres en mémoire, puis à l'aide de ces informations, charge le Master Boot Record qui contient toutes les informations sur le système de fichiers, ici FAT32. On obtient ainsi toutes les informations nécessaires à la lecture et écriture de fichiers (taille des clusters, root cluster, adresse des FAT, ...).
- `init_root` : Rajoute les liens . et .. à la racine, car ils n'existent pas par défaut ! Cela récupère aussi des informations sur la racine, pour pouvoir ouvrir directement ce répertoire.
- `init_stderr` : Initialise les messages d'erreurs, et redirige l'affichage d'erreur et de debug du noyau dans le fichier `/error/stderr`.
- `init_filename_gen` : Initialise les paramètres de génération de noms courts pour le système de fichiers.

Une fois passée cette grand étape d'initialisation, on peut enfin rentrer dans le véritable fonctionnement de l'OS.

2 Gestion de la mémoire

2.1 Mémoire Physique (`paging.c`)

Pour la gestion de la mémoire physique, nous n'avons rien fait de compliqué : on maintient une table de bits initialisée dans le fichier `paging.c` qui indique si chaque frame (4096 octets) est utilisée ou non. Lorsqu'une nouvelle frame est demandée, on trouve la première accessible, en recherchant exhaustivement, on la marque occupée. On peut de même libérer des frames. Un algorithme aussi simple est légèrement coûteux, mais cela reste mineur si la mémoire n'est pas énormément occupée. On pourrait faire un algorithme plus perfectionné, mais celui-là nous a suffi.

2.2 Mémoire Virtuelle (`memory.c`, `lib/malloc.c`)

Un "véritable" OS fonctionne en Higher Half, c'est à dire est mappé dans l'espace virtuel à la fin de la mémoire, ici par exemple `0xC0000000`, i.e 3Go. Ainsi, les programmes exécutés peuvent être compilés pour l'adresse 0, car ils disposeront des basses adresses virtuelles. Dans notre cas, nous n'avons pas réussi à réaliser un linker qui permettait de faire fonctionner le noyau en Higher Half. En effet, la difficulté était de booter et d'installer le paging sans utilisées d'addresses relatives. Nous avons donc choisi de garder le noyau dans le bas de la mémoire, et de compiler nos programmes à l'adresse `0x40000000` (1Go), à l'aide d'un nouveau linker.

Pour ce qui est de la gestion de la mémoire au sein d'un programme, chaque processus dispose d'une pile de taille fixée, et d'un tas extensible, géré par les fonctions `malloc` et `free` situées dans le fichier `malloc.c` de la library user. De même, le noyau dispose d'un tas, de taille fixée cette fois, mais aussi des fonctions `malloc` et `free`, qui cette fois renvoient des adresses alignées sur `0x1000`, car cela sert uniquement pour les page directories, et les page tables.

Ainsi, chaque processus a un page directory contenant :

- Le noyau, de `0x00000000` à la fin du tas du noyau
- Le code du processus, et son segment de données, à `0x40000000`
- La pile, puis le tas du processus, situés à `0x80000000`
- L'écran virtuel, situé encore plus loin

3 Gestion des Fichiers (`fat32/*`)

Nous avons implémenté une interface avec le système de fichiers FAT32, qui contient toutes les opérations nécessaires à l'utilisation des fichiers, définies dans `fs_call.c`. Nous avons voulu conserver la notion de file descriptor, nous avons donc une table des fichiers ouverts, contenue dans le noyau. Dans cette table se trouvent les informations nécessaires à l'accès au fichier, il a donc fallu sauvegarder beaucoup plus d'informations qu'avec un système de fichiers de type EXT, car la notion d'inode n'existe pas. Les informations sur un fichier sont contenues dans les entrées du répertoire qui le contient, pour éviter de les y extraire à chaque opération sur le fichier, on sauvegarde donc tout cela dans la table des fichiers.

Nous n'avons pas eu le temps d'implémenter une table de fichiers propre à chaque processus (comme nous l'avons fait avec les channels, décrits ci-après), ainsi, un processus peut ouvrir plusieurs fois le même fichier, et accéder, s'il connaît le numéro du file descriptor, à n'importe quel autre fichier ouvert. Il suffirait de mémoriser les fichiers ouverts par chaque processus, ce qui permettrait de plus de les fermer si un processus oublie de le faire avant de mourir. Puisque cela n'est actuellement pas réaliser, si un processus oublie de fermer un fichier ouvert, celui-ci n'est jamais fermé. Cela peut donc conduire à l'atteinte du nombre maximal de fichiers supporté par notre OS (fixé arbitrairement à 256).

Dans le système de fichiers FAT32, il existe pour chaque fichier un nom court et éventuellement un nom ong. Pour simplifier, le nom court est généré à l'aide d'un compteur, tandis que le nom long contient le nom véritable.

Dans de nombreuses fonctions du système de fichiers, nous utilisons des buffers déclarés localement, et donc alloués sur la pile. On pourrait remplacer cela par des appels à `malloc`, mais puisque la taille de pile est en pratique suffisante, nous ne l'avons pas fait.

Nous avons aussi codé une interface permettant d'utiliser par exemple `fprintf` pour écrire dans un fichier. Cette interface écrit en fait dans un buffer de taille fixée (512 octets en pratique), et fait appel à la fonction `flush` lorsque celui-ci est plein. Cette fonction `flush` écrit vraiment dans le fichier. Cela est implémenté dans le fichier `stream.c` du noyau, et ne sert en pratique dans le noyau qu'à la sortie de debug citée précédemment : `/error/stderr`. Du côté

de l'utilisateur, nous avons ajouté à cette interface `stream` vers un fichier la possibilité de rediriger en fait vers un `channel`, ce qui permet d'utiliser les fonctions `fprintf` et la même interface dans les deux cas. Ceci est codé du côté de l'utilisateur dans `lib/stream.c`.

4 Gestion des Interruptions

- Les interruptions correspondant à des erreurs sont traitées dans `isr_handler` dans le fichier `isr.c`. Si l'erreur nécessite l'arrêt du processus, celui-ci est tué, et on réordonne les processus.
- L'interruption clavier, elle, complète un buffer d'événements, qui peuvent être récupérés par le processus affiché à l'écran en faisant l'appel système `get_key_event`. La touche "2" est traitée à ce niveau et permet de changer le processus affiché à l'écran, ainsi que la touche "`*`", qui provoque l'arrêt du noyau et un aperçu de son état.
- L'interruption timer incrémente le temps de l'OS, réduit d'une tranche de temps le compteur du processus courant, et si besoin élit un nouveau processus.
- L'interruption `0x80`, qui correspond aux appels systèmes, appelle le handler associé au numéro de l'appel passé dans le registre `eax`, qui s'occupe de décoder les registres et d'effectuer l'appel souhaité.

Lorsque ces appels finissent, on utilise `iret` pour reprendre l'exécution du processus courant, ou du nouveau processus. Nous n'avons pas implémenté un changement de pile, ce qui signifie que lors de ce passag en mode kernel, nous utilisons la pile du processus courant. En pratique, nous n'avons pas eu encore de problèmes avec cela, mais une amélioration souhaitable serait de passer sur une pile kernel. En effet, dans le cas où le registre `esp` du processus est corrompu, alors cela conduit à un crash du kernel, qui tente de l'utiliser.

Nous ne changons pas de page directory lors de ces appels, sauf lors d'un changement de processus, car le noyau est mappé dans tous les page directories.

Nous ne passons pas réellement en user mode, car nous n'avons pas eu le temps d'implémenter le changement de segments et de droits. Toutefois, lors des appels systèmes, les pointeurs passés en arguments sont vérifiés, et rejetés s'ils pointent vers des zones de privilège 0 (kernel).

Durant tous les passages en mode kernel, les interruptions sont désactivées. Cela a l'avantage d'être sûr de ne pas être interrompu par un autre événement timer durant un appel système, mais cela signifie que les ticks du timer qui ont lieu pendant un appel système sont oubliés. En pratique, un processus qui fait un appel système en boucle passe donc la très grande partie de son temps en mode kernel, ce qui empêche le timer d'arriver. Nous avons donc rajouté une pénalité des appels systèmes sur le compteur des processus pour éviter une telle famine. En revanche, il faudrait, avec plus de temps, rétablir les interruptions en mode kernel, afin de pouvoir réveiller des programmes endormis dans ce cas là.

5 Gestion des Processus

5.1 Fonctionnement global (`kernel.c`)

La gestion des processus est basée sur le micronoyau, sur lequel nous avons greffé tout le reste de notre OS. Ainsi, les processus ont chacun une priorité (en pratique, tous les processus créés sont de priorité maximale, nous n'avons pas encore mis de paramètres pour cela), et peuvent se trouver dans chacun des états suivants :

- `FREE` : Ce processus n'existe pas
- `RUNNABLE` : Ce processus est prêt à être exécuté
- `SLEEPING` : Ce processus est dans la file des processus ayant fait l'appel système `sleep`, et sera réveillé après un certain nombre d'événements timer
- `BLOCKEDWRITING` : Ce processus est bloqué en attente sur un `channel` pour écrire
- `BLOCKEDREADING` : Idem, en lecture
- `WAITING` : Ce processus attend qu'un de ses fils meure
- `ZOMBIE` : Ce processus est mort, et attend que son père récupère la valeur d'`exit`

Nous avons gardé le fonctionnement du micronoyau : lorsqu'un processus voit son compteur arriver à 0, il est remis à la fin de sa file, et le prochain processus `RUNNABLE` est exécuté à la place. Pour les processus endormis, nous gardons une liste triée (file) avec le temps supplémentaire à attendre pour chaque processus. Lorque le premier arrive à 0, il est libéré, et ainsi de suite. Le processus 0, nommé `scavanger`, s'occupe d'éliminer les processus `ZOMBIE` orphelins.

Concernant le fonctionnement des processus, nous avons les appels systèmes suivants :

- `exec` : Crée un nouveau processus avec le fichier de code donné, et l'argument donné, et les `channels` d'entrée et sortie spécifiés
- `wait` : Attend la mort d'un fils
- `exit` : Se suicide
- `sleep` : S'endort pour la durée indiquée de ticks du timer, donc ici en millisecondes
- `kill` : tue le processus donné (nous n'avons pas mis de droits quelconques sur cet appel système)

En plus de cela, un processus à la fois dispose du "focus", et son écran virtuel est vraiment affiché. On peut changer de processus à l'aide de la touche "2". Le processus en focus est le seul à pouvoir recevoir l'entrée du clavier, via l'appel système `get_key_event`.

Les processus disposent de plus de tous les appels systèmes liés au système de fichiers, ainsi que pour modifier la taille du tas, et accéder à la date (cf `lib/syscall.c`).

Lorsqu'aucun processus n'est en état `RUNNABLE`, la variable globale `no_process` est mise à 1, les interruptions sont rétablies. Le noyau exécute un certain nombre de `hlt`, puis essaie de `reorder`, et ce jusqu'à l'infini. Dans ce cadre, les interruptions retournent directement après avoir mis à jour les processus endormis, sans provoquer d'autres modifications. (En étendant cette situation à tous les appels systèmes, cela permettrait de rétablir les interruptions en mode kernel).

5.2 Communication entre processus (`channel.c`, `lib/stream.c`)

Pour communiquer entre eux, les processus disposent de `channels` de taille fixée (512 octets). Comme annoncé précédemment, l'interface implémentée dans `lib/stream.c` permet aux processus d'user de la fonction `fprintf` pour écrire dans un `channel`, en passant par un buffer de taille lui aussi fixé, et de opérations de `flush`.

Le numéro de `channel` est identique à celui utilisé en tant que `stream`, ce qui, du point de vue de l'utilisateur, permet de confondre les deux. Cela permet d'éviter de multiples appels systèmes pour chaque caractère lu ou écrit, en groupant à travers le buffer du `stream`.

Ainsi, 4 appels systèmes permettent d'utiliser les `channels`:

- `new_channel` : crée un nouveau `channel` accesible en écriture et lecture
- `send` : appel non bloquant permettant d'envoyer un certain nombre d'octets dans le `channel`
- `receive` : appel non bloquant permettant de recevoir un certain nombre d'octets du `channel`
- `wait_channel` : appel qui, selon les arguments, vérifie si l'écritue est possible sur un `channel` , ou la lecture, et bloque jusqu'à ce que ce soit possible. Cela ne bloque pas si le processus est seul sur le `channel` .

Nous avons choisi de séparer de cette façon appels bloquants et non bloquants pour simplifier le côté du kernel, car il était complexe d'accéder aux données contenues dans la mémoire du processus qui n'est pas courant, du fait que l'on utilise la pile du processus courant. Puisqu'un processus ne peut de plus attendre que sur un `channel` , il était donc nécessaire d'avoir des appels non bloquants lorsque le processus fait face à plusieurs `channels`.

Ainsi, l'entrée standard (`channel` 0) et la sortie standard (`channel` 1) sont des `channels` de ce type. Contrairement à la table de fichiers, le noyau stocke pour chaque processus la liste de ses `channels`, ce qui permet de savoir quand un processus est seul sur un `channel` , et d'éviter que les uns accèdent aux données des autres.

Nous ne sommes pas attardés sur la sortie d'erreur, même si le code des programmes écrits utilise cette sortie. Ainsi, nous avons fixé pour l'instant `STDERR` = `STDOUT`. Une amélioration possible et peu coûteuse serait de distinguer potentiellement ces sorties, en rajoutant un paramètre à `exec`.

Nous avons enfin mis sur ces channels des droits, c'est à dire que l'entrée standard n'est accessible qu'en lecture, la sortie qu'en écriture.

6 Exemples de Processus (`programs/src/*`)

Muni de tous ces outils, nous avons écrits divers programmes exécutables sur notre OS.

6.1 Console

La console est un programme qui gère un écran, l'entré du clavier, et communique avec un programme donné. Elle s'occupe de gérer le curseur, d'afficher les caractères reçus par le programme à l'écran, et transmet l'entré clavier filtrée au programme concerné.

Nous avons réinventé nos propres séquences pour que le programme puisse changer les couleurs, la position du curseur, et d'autres paramètres de la console, ce qui correspond à la séquence ESC + "[", suivie des commandes, et terminée par "|". Tout cela est définit dans `lib/printing.h` et accessible via la fonction `esc_seq`.

Une console se suicide lorsqu'elle se retrouve seule sur un ses `channels` de communication.

6.2 Shell

Nous avons codé un shell minimal, avec les fonctions de base, à ce jour les suivantes : `ls`, `mkdir`, `rmdir`, `cp`, `rm`, `mv`, `ps`, `pwd`, `cat`, `touch`, `kill`, `color`, `clear`, `fwrite` et enfin `splash` et `cacatoes`.

Nous avons implémenté un simple historique des commandes, et la possibilité d'utiliser des pipe : "`|`", des redirections "`>`", "`>>`" et "`<`", ainsi que l'exécution en arrière plan "`&`".

Les flèches verticales permettent de naviguer dans l'historique, et celles horizontales permettent de faire défiler le texte, vers le heut ou le bas.

7 Limitations, Difficultés et Améliorations

- Nous ne tenons pas compte de la taille de la mémoire, nous supposons que nous avons 4 Go. Il suffirait de lire la structure multiboot, et d'en tenir compte
- Un certain progrès serait de passer en Higher Half
- Chargement des programmes : nous compilons vers des simples binaires, et nous allouons une taille constante pour le code et le segment de données. Une amélioration simple serait d'utiliser, comme les autres groupes des fichiers `.elf`, que l'on parserait
- Rétablir les interruptions pendant le mode kernel, pour ne pas ratet des ticks du timer
- Utiliser plus d'allocation dynamique pour s'affranchir des allocations statiques bornées, par exemple pour les channels. Le fichier `kernel.c` a été écrit au début, sans malloc, et n'utilise donc que des fonctions d'allocation dédiées dans des tableaux statiques

- Si un processus est en état `ZOMBIE` et que son père meurt, si `scavanger` est déjà en attente, ce dernier n'est pas mis à jour. Des `ZOMBIES` orphelins peuvent donc temporairement (jusqu'au prochain `wait` de `scavenger`) persister
- Implémenter une table de fichiers ouverts pour chaque processus
- Ouvrir les répertoires comme des sortes de `stream` en lecture, pour éviter de relire le disque pour chaque entrée, mais lire par blocs et sauvegarder les blocs lus jusqu'à consommation. (Un peu comme ce que fait le programme `cat`)

8 Compilation et exécution de CacatOS

Pour compiler et exécuter CacatOS, il faut en particulier créer une image de disque dur contenant une partition FAT32, installer GRUB dessus, puis synchroniser l'OS compilé, ainsi que les autres programmes compilés, à la racine de cette partition. Cela utilise donc les outils préexistants, et demande de monter l'image du disque pour synchroniser facilement. On suppose que le dossier `/mnt/test` est libre. Il y a donc besoin de permissions pour monter cette image.

La commande `make` est supposée créer tout cela, et lancer QEMU sur cette image disque. En cas de problème de compilation / création du disque, une image vierge du disque avec seulement GRUB d'installé est disponible sur notre Git `https://github.com/VengeurK/SysResProj` (`resources/disk.img`). De même, il y a une image du disque avec déjà tous les fichiers compilés synchronisés dessus : `resources/disk_ready.img`. Il suffit de copier l'image souhaitée dans `resources/`, et ensuite relancer `make`.