Croisière au cœur d'un OS*

Résumé

Après un survol à la frontière du noyau Linux (hors-séries 16 et 17), nous vous proposons une croisière au long cours, une sorte de mode d'emploi de l'*OS dont vous êtes le héros*. Cet article est en effet le premier d'une série d'une dizaine d'articles, qui vise à présenter des outils, les concepts principaux, et, bien sûr, du code pour faire son propre noyau de système d'exploitation. Au programme de ce premier article : booter et afficher un message. C'est parti !

Introduction

Dans cette série d'articles, nous allons tenter de montrer que l'écriture d'un système d'exploitation (OS) est certes "difficile", mais pas "inaccessible" aux communs des informaticiens.

Pour cela, nous vous proposons non pas de *disséquer* un OS existant, si petit soit-il, pour *illustrer* les concepts fondamentaux les uns indépendamment des autres. Nous vous proposons plutôt de rentrer dans le vif du sujet par vous-mêmes : "c'est en faisant qu'on apprend". Ce sera l'occasion d'appréhender concrètement que la difficulté de réalisation d'un OS ne réside pas fondamentalement dans la réalisation des sous-systèmes indépendamment (systèmes de fichiers, gestion des espaces d'adressage, ordonnancement, pilotes de périphériques, ...). Mais que la difficulté réside dans la complexité de l'*intégration* de tous ces sous-systèmes ensemble, et dans le perpétuel effort pour atteindre un Graal dans l'élégance de cette intégration.

Principe de la série

Pour que cette croisière ne devienne pas une galère, nous ne nous contentons pas de fournir une recette de cuisine et de vous laisser vous débrouiller. Chaque article s'articule autour d'un concept et de fonctionnalités particulières, et s'accompagne du code les implantant. Le code de l'OS complet se construit ainsi de façon incrémentale : d'un article sur l'autre nous fournirons le *patch* résumant les modifications du code ; nous fournirons aussi l'intégralité du code à chaque fois. À chaque numéro, l'OS compilera, et pourra être lancé (avec grub ou un secteur de boot fourni dans un premier temps) sur vraie machine, ou dans un émulateur (bochs ou qemu) : c'est le rôle de la petite *démonstration* concluant chaque article, qui doit vous inciter à la modifier pour tester par vous-même.

Objectifs de la série

Autant insister tout de suite : le but n'est pas de concurrencer les OS existants, mais d'apprendre de façon progressive ce qu'il y a dedans. Certes, les fonctionnalités implantées sont celles qui permettent de décrire les concepts fondamentaux intervenant dans les OS modernes. Mais elles ne sont pas abouties à la hauteur des fonctionnalités exigées dans les OS dignes de ce nom (Linux, Solaris par exemple, et même... Windows). De même pour les *démonstrations* proposées qui resteront extrêmement basiques.

Nous avons dû en effet faire des choix et des compromis sur les fonctionnalités montrées, et la manière dont nous les avons implantées, dans le sens d'une meilleure lisibilité et compréhension. Et ceci sans occulter les détails purement techniques souvent bloquants pour le débutant, bien que totalement dépourvus d'intérêt pour l'expert.

Pour tout cela, nous avons supposé que vous disposez :

- d'une bonne maîtrise de la programmation en C (outils, notions classiques, et notions plus bas niveau telles que les opérations binaires, et autres masques de bits),
- de quelques notions d'assembleur (x86 si possible),
- de rudiments sur les concepts fondamentaux d'un OS, même si ces notions seront redécouvertes dans la série petit à petit. Pour le novice, ou pour réviser, voir les 2 hors-séries sur le noyau Linux (16 et 17).

Si l'assembleur vous fait peur, sachez que des rudiments sont suffisants. Nous nous en satisfaisons, et d'ailleurs les experts du domaine ne manqueront certainement pas de relever nos insuffisances sur le sujet.

Pour le reste, en plus des explications et du code, nous essaierons de donner des références de documentations pertinentes.

Avant d'embarquer...

Sachez que la version des sources pour l'article du mois en cours est fournie dans le CDROM du magazine, et que les sources pour les mois précédents seront sur `http://sos.enix.org`. Une mailing-list est également mise en place pour les éventuelles questions, suggestions, rapports de bugs, etc... Toutes les informations sont disponibles sur le site précédent.

*La version originale de cet article a été publiée dans GNU Linux Magazine France numéro 62 – Juin 2004 (`http://www.linuxmag-france.org`) et cette version est diffusée avec l'autorisation de l'éditeur.

Enfin, avant de démarrer cette série, nous tenons à signaler que ces articles n'auraient pas été possibles sans la base que nous avons acquise par et pour le projet KOS ("Kid Operating System", `http://kos.enix.org`). Ce projet, à but d'apprentissage, nous permet d'apprendre par la pratique, d'expérimenter en grandeur nature, et de tester de nouveaux concepts. SOS s'inspire de KOS pour une grande partie.

Merci donc à tous les membres qui sont, ou qui ont été actifs au sein de Kos. Merci tout particulièrement à Thomas Petazzoni, le coordinateur de KOS, qui a établi le plan de la série, et qui, je l'espère très vivement, devrait me rejoindre pour les prochains articles. Et merci bien sûr à tous les OS libres existants, en particulier Linux et les BSD, à Grub, à Bochs, aux outils GNU sur lesquels nous reposons ici très largement, et aux formidables livres traitant du sujet [1, 2, 3].

1 Présentation de l'OS

Comme tout système d'exploitation, celui que nous vous proposons vise les objectifs fondamentaux suivants :

- Faire l'interface entre les demandeurs (applications dites utilisateur) en ressources (espace mémoire, fichiers, connexions réseaux, ...), et les ressources disponibles : processeurs, mémoire, disques, carte réseau, ...
- Gérer les offres et les demandes dans leur ensemble, pour *arbitrer* leur attribution dans l'espace (allocation), et dans le temps (ordonnancement), de manière à optimiser un critère donné, tel que la réactivité moyenne, le temps de réaction pire cas, le nombre d'applications chargées à un instant donné, ...

Nous ne nous étendons pas sur ces généralités ici, préférant rentrer dans le vif du sujet.

1.1 Caractéristiques de SOS

"SOS" est son nom, pour "Simple Operating System", ou "OS court" en français dans le texte, et sans aucune référence au "Sophisticated OS" d'Apple. Nous donnons ci-dessous ses caractéristiques et limitations, avec quelques références techniques obscures qui seront approfondies au fur et à mesure des articles :

- Architecture cible : monoprocesseur Intel 80486 ou supérieur (dite "`x86`" par la suite), type PC, non pas par élégance technique, mais parce que cette architecture est sans doute la plus répandue
- Machine hôte pour la compilation : toute machine de type Unix. Pour l'exécution : par exécution réelle ou simulation (bochs/qemu) sur PC, par simulation seulement sur les autres machines
- Noyau de type monolithique, interruptible, non préemptible ("*big kernel lock*")
- Chargement par un chargeur de boot compatible avec le standard multiboot [4] (Grub pour le moment), ou par un secteur de boot fourni et valable jusqu'à l'article 2 seulement
- Gestion des interruptions matérielles (PIC i8259, pas d'APIC), logicielles, et des exceptions processeur, gestion des ports d'I/O x86
- Séparation des notions d'adresses physiques et d'adresses noyau
- Allocateur dans la zone noyau ségrégationniste de type *slab*
- Pas de *swap*, pas de *reverse mapping*
- Périphériques : clavier, zone vidéo du PC, disques IDE de bochs (et des PC pas trop exotiques)
- Autres pilotes : partitions PC, VFS (*Virtual File System*) basique et système de fichiers FAT, points de montage définis "en dur" (à la DOS). Pas de pile de protocoles réseau
- Support des applications utilisateur : programmes au format ELF32, sans bibliothèques partagées, notion de processus et de *threads*, libC minimale avec `read()`, `write()`, etc... et `mmap()`.

Ces caractéristiques risquent d'évoluer au fil des articles, mais dans le sens de leur amélioration. Par ailleurs, pour toute suggestion de fonctionnalité, nous vous invitons très vivement à nous proposer les patches les implantant. De même, nous signaler tout bug, c'est très important !

1.2 Plan prévisionnel

Plutôt que de détailler les objectifs de cet article, nous préférons le situer dans la série au travers du schéma 1, qui indique les concepts fondamentaux que nous comptons décrire.

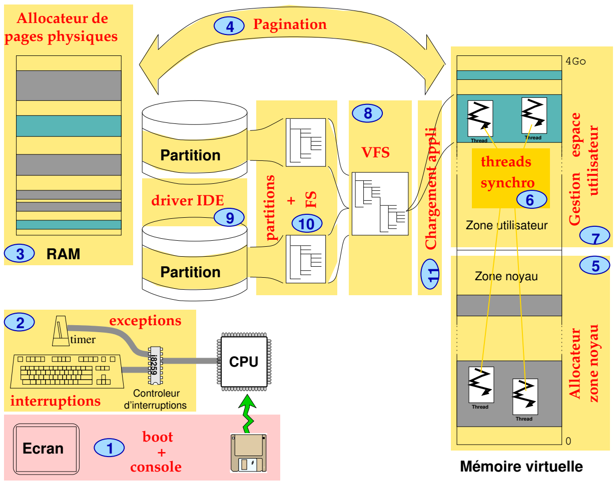


FIG. 1 – Programme des articles

Comme on peut s'en rendre compte, le présent article occupe une place négligeable dans ce schéma. C'est qu'il est atypique : l'introduction et la présentation de SOS y sont centrales, et son objectif technique n'est pas uniquement interne à l'OS. En effet, nous comptons dans la suite de l'article 1/ donner quelques informations techniques basiques sur l'architecture cible, 2/ présenter les outils et les méthodes de génération de l'OS, 3/ détailler la phase de chargement de l'OS (grub et secteur de boot), et 4/ donner quelques compléments d'information relatifs à l'édition de liens (en annexe). Paradoxalement, cet article sera vraisemblablement le plus long.

2 Élements d'architecture

2.1 Au cœur se trouve le processeur

C'est bien sûr le processeur qui est chargé d'exécuter les instructions les unes à la suite des autres. Il va chercher ces instructions en mémoire, puis les exécute, ce qui se traduit par des opérations élémentaires sur des données. Les données peuvent être de 2 types. Soit ce sont des entiers (ou des flottants, mais nous n'en parlons pas ici) écrits par le programmeur ou générés par le compilateur, et qui correspondent soit à des valeurs numériques, soit à des adresses mémoire. Soit ce sont des registres.

Les *registres* sont des petites zones de mémoire internes au processeur, destinées à rassembler 1 seul entier, ou 1 seul flottant, ou 1 seule adresse. Ils sont d'accès très rapides puisqu'ils sont potentiellement accédés une ou plusieurs fois par cycle d'horloge du processeur (plusieurs GHz de nos jours). Mais d'une part ils ont une capacité très limitée (1 entier, ou 1 flottant, ou 1 adresse), et d'autre part sont en nombre très limité. Le processeur du PC est un des moins bien dotés sur ce plan : il y en a une vingtaine [5, section 3.2], pas davantage ! On donne un nom à chacun d'eux : par exemple `eax`, `ebx`, `ecx` et `edx` sont les registres pour stocker des entiers ou des adresses mémoire quelconques sur 32 bits. En plus de ces registres dits *généraux*, il y a notamment les registres suivants :

Le pointeur courant (`eip`) : il s'agit du registre dans lequel est stockée l'adresse de l'instruction que le processeur exécute. Ce registre est mis à jour par le processeur lui-même lorsqu'il suit la séquence d'instructions, ou lorsqu'il rencontre une instruction de saut (appel de fonction, condition du type `if... then... else...`),

Le pointeur de pile (`esp`) : l'adresse du "sommet de la pile" courante. Nous ne nous étalerons pas sur le concept de pile maintenant. Nous dirons seulement qu'il s'agit d'une portion de la mémoire destinée à contenir les variables locales aux fonctions, les paramètres lors des appels de fonctions, et les adresses auxquelles les fonctions appelées doivent retourner pour redonner la main aux fonctions appelantes. Le registre `esp` est mis à jour directement par le processeur quand il rencontre les instructions de manipulation de pile (du type `push`, `pop`), ou des appels de fonctions (du types `call` et `ret`).

Il existe d'autres registres dont nous reparlerons plus tard. Sur les processeurs de la famille `8086`, tous ces registres étaient à l'origine d'une largeur de 16 bits, et ont été agrandis à 32 bits lors du passage au `80386`.

2.2 Il est relié au reste par des bus...

Du point de vue de l'électronicien, un "bus" n'est ni plus ni moins que x fils qui vont d'une puce à plusieurs autres puces sur la carte mère, les fils faisant transiter des bits en parallèle. Pour x fils, on parle d'une "largeur de bus" de x bits. En informatique, on dira "un bus" sachant qu'on parle en fait de 2 bus au sens de l'électronicien : le bus d'*adresses* et le bus de *données*, en référence au fait que pour accéder à une donnée en mémoire, il faut fournir une adresse. Sur un ordinateur, il existe plusieurs bus externes au processeur [6], et donc plusieurs manières de dialoguer avec les périphériques qui y sont connectés. La figure 2 résume l'architecture canonique de ces bus.

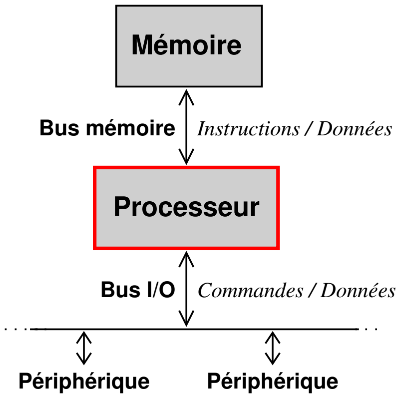


FIG. 2 – Organisation des bus

Bus mémoire. Pour relier le processeur à la mémoire physique (notamment la RAM), on parle bien évidemment du "bus mémoire", qui couvre l'*espace des adresses physiques*. Pour y accéder sur un PC, le compilateur génère des instructions de la famille `mov`, ou utilise le fait que la plupart des instructions admettent des adresses comme paramètres (`add`, `inc`, ...).

Sur `80386`, la largeur des adresses du bus mémoire est de 32 bits (36 bits avec l'extension PAE, pour *Page address extension*, à partir du Pentium Pro), celle des données peut varier de 32 à 128 bits.

Bus d'entrée / sortie. Sur la plupart des architectures [6], ce bus relie le processeur aux périphériques autres que la mémoire, tels que les autres bus (PCI, AGP, ISA, IDE, ...). Pour y accéder, on doit utiliser explicitement des instructions spéciales fournies par le processeur, ou admettre que telle plage connue d'adresses physiques manipulées par le processeur est déroutée sur ce bus, alors que le reste des accès se fait sur le bus mémoire.

Cependant, sur l'architecture PC, ce bus se confond avec le bus mémoire vu ci-dessus, même si ils ne fonctionnent pas à la même vitesse ou n'ont pas la même largeur de données. C'est le rôle des *chipsets* des PC actuels que d'assurer l'interconnexion de ces bus avec le processeur, de sorte que ces deux bus distincts "physiquement" n'en forment qu'un seul "logique" du point de vue du programmeur et du processeur (voir la figure 3). Dans toute la suite, nous confondrons ces

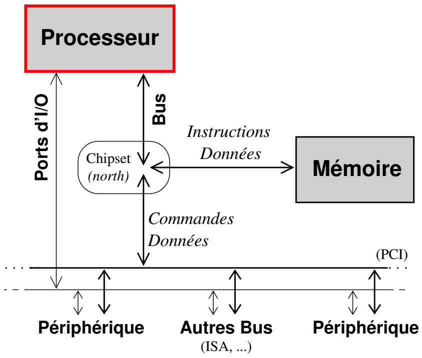


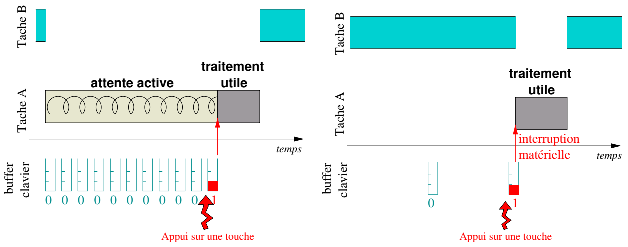
FIG. 3 – Organisation des bus sur le PC

deux bus sous le terme générique de "bus mémoire", en sachant que, sur d'autres architectures, les modes d'accès à ces deux bus sont totalement différents.

Donc sur PC, lorsqu'on accède à une adresse a en mémoire, on tombe soit 1/ sur la mémoire physique ou le Bios (adresses basses, *ie* a petit), soit 2/ sur un périphérique matériel, soit 3/ sur rien du tout (comportement indéterminé). Si une adresse sur le bus mémoire permet d'accéder à un périphérique, on dit que ce périphérique est "projeté" ou "*mappé*" en mémoire physique. Les adresses des périphériques ainsi mappés sont définies par le matériel même, ou par le Bios. Nous verrons dans la section 4.3.3 un exemple de périphérique mappé en mémoire physique.

Ports d'entrée / sortie. Sur le PC, il existe un autre bus [5, chapitre 13], qui permet d'accéder aux "*ports d'entrée/sortie*", et qui est de largeur plus réduite que les précédents (adresses sur 16 bits). Sur ce bus, chaque adresse s'appelle un *port*. Le processeur utilise 2 familles d'instructions spécialisées pour y accéder, que le compilateur ne génère pas spontanément : `inb` (données 8 bits), `inw` (données 16 bits), `inl` (données 32 bits) pour la lecture du bus, et `outb/outw/outl` pour l'écriture sur le bus. La différence (assez subtile) entre ce bus et le *bus d'entrée/sortie* précédent, est que ce bus ne sert à accéder qu'aux *vieux* périphériques, *ie* ceux qui forment le standard PC depuis le début (contrôleur d'interruptions, bus ISA, ...). Les périphériques plus récents sont mappés sur le bus d'entrée/sortie ci-dessus, qui se confond avec le bus mémoire sur PC.

L'utilisation de ce bus tend à être déconseillée. D'une part parce que cet espace est relativement réduit. Et d'autre part parce que ce mode d'accès est spécifique aux "vieilles" familles de processeurs, dont le `i80x86` fait partie [7]. Cependant, le poids de l'histoire faisant, on le retrouve encore dans le dialogue avec la plupart des périphériques propres au PC (contrôleur d'interruptions, clavier, contrôleur IDE, ...).



(a) Attente active (b) Interruption matérielle

FIG. 4 – Modes de signalisation d'un événement matériel

2.3 ... et il est aussi relié au reste par les interruptions

En termes électroniques, une *interruption* est quelque chose de beaucoup plus simple qu'un bus. Il faut se représenter une *ligne d'interruption* comme étant un simple fil qui relie un périphérique (contrôleur clavier, carte réseau) au processeur. Le rôle de ce fil est d'être dans un état (0 ou 1) quasiment toujours identique **sauf** quand quelque chose de remarquable se produit : l'utilisateur vient d'appuyer sur une touche, un paquet réseau vient d'arriver, etc...

Lorsque le processeur détecte que la ligne d'interruption change d'état, il *interrompt* le programme qui est en train d'être exécuté, et fait un saut vers la première instruction d'un petit programme, ou *routine*, destiné à effectuer un *traitement* associé à cette *interruption*. Comme par exemple interroger le contrôleur clavier pour connaître la/les touche(s) qui a/ont été enfoncée(s) (figure 4), ou interroger la carte réseau pour stocker le/les paquets reçus en mémoire, etc... avant de retourner dans le flot d'instructions qui avait été interrompu.

Ce type de traitement améliore les performances du système au détriment d'une légère perte en réactivité. Cela permet en effet de prévenir le système d'exploitation de la venue d'un évènement, sans qu'il soit nécessaire pour lui de faire une *attente active*, ou *polling* (figure 4(a)), pour demander en permanence au clavier ou à la carte réseau si un évènement est arrivé : pendant ce temps, aucun autre traitement plus utile ne peut être exécuté. Une interruption interrompt donc le flot d'instructions sans préavis (figure 4(b)) : son fonctionnement est *asynchrone*, et autorise d'autres traitements à être exécutés en attendant la survenue de l'interruption. En fait, pour ce type d'interruption, on préfère parler d'*interruption matérielle*, car nous verrons que le vocable d'*interruption* s'applique aussi à d'autres moyens de signalisation à l'OS, qui ne sont ni d'origine matérielle, ni asynchrones.

Sur le PC originel, il n'y a que 16 lignes d'interruptions matérielles concentrées par deux *contrôleurs* d'interruptions (ou *PIC* pour *Programmable Interrupt Controller*) vers le processeur, afin de n'utiliser que 2 pattes supplémentaires au niveau du processeur. Ces contrôleurs sont branchés en cascade, ce qui ne laisse que 15 lignes d'interruptions matérielles réellement utilisables par les périphériques. Ce nombre est petit, on est donc parfois amené à brancher plusieurs périphériques sur la même ligne d'interruption : il y a alors *partage* d'interruption ; il y a encore peu de temps on disait qu'il y avait *conflit* d'interruption.

Sur les machines modernes, on peut s'affranchir de cette organisation restrictive : le processeur peut jouer le rôle des contrôleurs d'interruption (on parle d'*APIC* pour *Advanced* PIC), et gérer jusqu'à 256 lignes. La compatibilité avec l'ancienne organisation à 2 contrôleurs en cascade est cependant conservée ; c'est d'ailleurs cette ancienne organisation que SOS utilisera.

3 Mise en place de l'environnement de travail

Nous supposons que vous disposez d'un système de type Unix. Cela recouvre Linux, MacOS X, Solaris bien sûr, mais aussi Windows avec Cygwin. En fait, le minimum vital est d'avoir un (cross-)compilateur pour l'architecture `i586`, et les `mtools`. Avoir le simulateur de PC bochs ou qemu est un très bon atout. Pour les PC, on s'arrangera aussi pour installer Grub, mais ce n'est pas obligatoire.

3.1 Environnement de compilation

Sur les PC. Sur les PC, gcc est en général disponible d'office. Pour SOS, nous avons testé et approuvé avec gcc 3.3.3 (debian) et gcc 3.2 (redhat 9). En principe, toute version de gcc de moins de 3 ans devrait convenir.

Sur les autres architectures. SOS ne fonctionne que sur les machines de type PC (intel `80486` ou supérieur), mais il est possible de le compiler à partir d'autres architectures grâce à un *cross-compilateur*. Pour en fabriquer un, on compilera d'abord les *GNU binutils* pour l'architecture `i586-gnu` (`./configure i586-pc-gnu` avec éventuellement un `--prefix=...`) : la dernière version est recommandée. Puis on compilera gcc à l'aide des options de configuration suivantes : `CFLAGS="-O2 -Dinhibit_libc" ./configure --target=i586-gnu --with-as=/chemin/vers/i586-gnu-as --with-ld=/chemin/vers/i586-gnu-ld --with-gnu-as --with-gnu-ld --enable-languages=c --disable-shared --disable-multilib --disable-nls --enable-threads=single` (avec éventuellement un `--prefix=...`).

Nous reposons sur des extensions de gcc, donc d'autres compilateurs ne conviendraient pas.

Quelques remarques. Nous n'avons pas besoin d'autres outils pour compiler SOS. Ainsi, contrairement à Linux, nous n'utiliserons pas l'assembleur `nasm` : nous utiliserons directement gcc qui appelle l'assembleur `gas` des binutils, même pour assembler le code 16 bits du secteur de boot fourni. Au passage, les habitués de la syntaxe d'assembleur Intel risquent d'être désarçonnés : `gas` utilise en effet la syntaxe ATT, qui (entre autres) inverse les paramètres des mnémoniques par rapport à la syntaxe Intel. D'autre part, les habitués de `nasm` risquent d'être également surpris par les macros qu'on retrouve aussi dans les sources assembleur (fichiers d'extension "`.S`") : c'est qu'avant d'appeler l'assembleur, gcc fait traiter le source par le préprocesseur `cpp` qui s'occupe de ces macros.

3.2 Génération de l'image de disquette

Dans SOS, nous commençons par générer un fichier "image", qui correspond à ce que contiendrait une vraie disquette. Cette façon de faire simplifie le test dans un émulateur (bochs ou qemu), et n'interdit pas de transférer cette image sur une vraie disquette. Dans l'absolu, on aurait cependant pu nous en passer.

Les `mtools`. Pour générer les images, nous utilisons les `mtools` [8], une série d'outils pour accéder aux disquettes et partitions msdos. Ils peuvent aussi accéder à des images de disques ou de disquettes. Sur les machines Linux, nous aurions certes pu préférer travailler avec des images au format `ext2` (`mount -o loop ...`). Mais les `mtools` sont plus simples à fabriquer ou à utiliser sur d'autres architectures ou systèmes, comme Windows ou Solaris par exemple. Normalement les `mtools` sont fournis avec toutes les distributions Linux. Sinon, leur compilation ne devrait pas poser de problème majeur.

Le chargeur de boot `Grub`. Á part pour les deux premiers articles, nous avons en plus besoin d'installer Grub sur l'image de la disquette. Grub [9] est un *chargeur de boot* comme Lilo. C'est-à-dire un programme dont le rôle est, dès l'allumage de la machine, d'aller chercher un noyau sur le disque, de le charger en mémoire, et de le démarrer. Nous entrerons dans les détails dans la section 4.2.2.

Pour installer Grub sur l'image de la disquette, nous proposons deux possibilités : 1/ soit vous installez Grub sur votre machine, ce qui fabriquera le programme `/sbin/grub` ou `/usr/sbin/grub` qui permettra d'installer un autre Grub dans l'image de la disquette. 2/ Soit vous ne l'installez pas sur votre machine, et dans ce cas nous fournissons une image de disquette avec Grub préinstallé (voir la section 4.4.2). Les distributions Linux sur PC proposent un Grub la plupart du temps, mais en installer un récent est toujours recommandé, notamment pour tester avec qemu (voir ci-dessous). Lors de la compilation de Grub, il est possible de lui préciser de supporter une carte réseau : il peut par conséquent charger le noyau par le réseau depuis un serveur distant (en utilisant `tftp`), ce qui est pratique lorsqu'on développe et teste sur deux machines différentes.

Sur les architectures autres que PC, il n'est pas possible d'installer Grub, donc la deuxième solution s'impose.

3.3 Environnement de test

Dans la section 4.4, nous indiquerons la marche à suivre pour tester. Ici, nous donnons quelques indications pour mettre en place les outils de test par simulation. Pour tester grandeur nature, nous ne détaillons rien, dans la mesure où il suffit de disposer d'un PC équipé d'un lecteur de disquettes et/ou d'un disque dur sur lequel Grub est installé (et peut éventuellement charger le noyau par le réseau).

Bochs [10]. Pour tester, la solution de choix est d'utiliser le simulateur bochs, qui a l'immense intérêt de simuler une machine de type PC sur pratiquement toutes les architectures. Il est disponible sur la plupart des distributions Linux. Lorsqu'il s'agit de voir fonctionner SOS, une telle version de bochs devrait être amplement suffisante. Mais si vous envisagez de faire des petites modifications de SOS pour tester par vous-mêmes, nous conseillons soit d'être sous debian testing/unstable et d'utiliser le bochs qui y est disponible (`apt-get install bochs-x vgabios`), soit de compiler votre propre bochs pour bénéficier de certaines fonctionnalités **très** utiles au debuggage. Ces fonctionnalités doivent être listées au moment du `./configure`.

En particulier, nous encourageons très fortement de configurer avec l'option `--enable-port-e9-hack` (adoptée par défaut dans le bochs de la debian) qui permet à l'OS simulé d'afficher des messages sur le terminal où bochs a été lancé (voir la section 4.4.1), rien qu'en écrivant sur le *port d'entrée/sortie 0xe9* de bochs. Ceci permet de garder une trace dans le terminal en cas de *crash*, de faire des copiers-collés, etc...

Pour les amateurs de `gdb`, il y a deux moyens de retrouver ses marques, c'est-à-dire de debugger l'OS comme si on debuggait un programme normal. La première méthode consiste à intégrer un debugger semblable à gdb directement dans bochs : ce sont les options de configuration `--enable-debugger --enable-disasm` qui le permettent (de paire avec `--enable-readline` pour bénéficier de l'historique). La deuxième méthode est d'utiliser gdb et sa commande `target remote hostname:port` : il faut pour cela configurer bochs avec l'option `--enable-gdb-stub`, mais ensuite il faudra systématiquement lancer gdb comme indiqué dans la section 4.4.1.

Qemu [11]. Il s'agit d'une alternative sérieuse à bochs, plusieurs dizaines de fois plus rapide, mais qui ne poursuit pas les mêmes objectifs. C'est un projet assez jeune, et très dynamique, vraiment très intéressant sur le plan technique. Avec une version récente de Grub, il est possible de tester SOS avec qemu. Même si certaines distributions Linux (debian unstable) proposent qemu, préférer quand même compiler vous-même la version cvs, car les développements se font à un rythme quotidien. La compilation ne devrait pas poser de problème majeur, mise à part qu'il faut avoir installé la bibliothèque `SDL` avant (sinon qemu compile, mais le PC simulé n'a pas d'écran). Par défaut, qemu ne propose pas de fonctionnalité analogue au `--enable-port-e9-hack` de bochs, mais nous fournissons le patch `extra/qemu-port-e9.diff` qui l'implante : il risque d'être nécessaire de l'adapter manuellement, compte tenu de la rapidité d'évolution de la version cvs de qemu.

4 Première version de l'OS : "Message in a bochs"

L'objectif technique de ce premier article est le suivant :

- Lors du boot : copie du noyau à partir de la disquette vers la mémoire, à l'adresse 0x200000 (*ie* 2Mo), définie dans `support/sos.lds` (voir l'annexe). C'est le rôle du *chargeur de boot*, c'est-à-dire de grub ou du secteur de boot fourni.
- La chargeur saute à l'adresse de la fonction `sos/main.c:sos_main()` : l'OS prend ainsi la main, et cette fonction est exécutée.
- Cette fonction en appelle quelques autres, pour afficher des messages à l'écran, mais également sur le port `0xe9` de bochs. Pour cela, des "*drivers*" et les services minimaux d'une bibliothèque C extrêmement basiques sont définis.
- L'OS ne sort **jamais** de `sos_main()` : il se termine par une boucle infinie.

Le figure 5 présente une copie d'écran une fois le lancement de SOS effectué... rien de très impressionnant en effet.

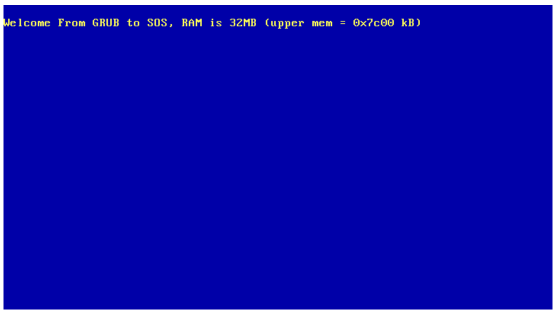


FIG. 5 – Copie d'écran après chargement du noyau

4.1 Arborescence des sources

Les sources de l'OS se répartissent principalement dans 4 répertoires :

`bootstrap/` : le code nécessaire pour que l'OS puisse être booté par Grub

`hwcore/` : le code de l'OS qui gère les éléments très bas niveau proches du processeur (contrôleur d'interruptions, vecteur d'interruptions, tables de traduction d'adresse, ports d'I/O)

`drivers/` : le code de l'OS responsable du contrôle des matériels supportés (mémoire vidéo x86, port `0xe9` pour la sortie de debuggage de bochs, contrôleur IDE de bochs, clavier)

`sos/` : le cœur de l'OS, qui intègre la gestion de tous les sous-systèmes (gestion de la mémoire virtuelle, ordonnancement et synchronisation, gestion des systèmes de fichiers).

Pour compiler le système, le répertoire `support/` fournit un fichier essentiel, responsable de la définition de l'organisation de l'OS en mémoire : le script `sos.lds` pour `ld`. Ce répertoire contient également un script shell chargé de construire une image de disquette en y installant Grub (`build_image.sh`).

Enfin, le répertoire `extra/` contient les fichiers pour compiler SOS avec un secteur de boot intégré, qui remplace Grub, mais qui ne sera valable que jusqu'à l'article 2. Il contient également le nécessaire pour compiler SOS avec Grub sur les architectures autres que x86 (SOS est développé principalement sur un powerpc).

4.2 Chargement de SOS

Cette première étape est prise en charge soit par Grub, soit par le secteur de boot fourni. Dans cette partie, nous allons détailler cette étape, qui souffre d'une lourdeur importante liée au poids considérable de l'Histoire de l'architecture cible. La phase d'initialisation d'un PC doit en effet être conforme en tous points au comportement originel qu'il y avait sur les PC de 1981 (!). Déjà alors le PC n'était pas un modèle d'élégance, autant dire que les couches d'architecture qui ont été rajoutées depuis n'arrangent rien. Ne parlons même pas de l'architecture x86-64 (ou x86e) qui rajoute encore une strate géologique.

4.2.1 Au début est le BIOS

Au démarrage de la machine, la mémoire vive ainsi que tous les périphériques matériels sont dans un état indéterminé. C'est donc un programme, stocké en mémoire morte (ROM), qui va prendre en charge l'initialisation de l'ordinateur et le démarrage d'un OS : le BIOS, pour *Basic Input/Output System* [12]. Cette ROM est mappée sur le bus mémoire entre les adresses `0xc0000` et `0xfffff` (*ie* 1Mo), et son contenu est écrit par le fabricant de la carte mère de la machine.

Lorsque la machine est initialisée, le processeur exécute sa première instruction à l'adresse physique `0xffff0` [13, section 9.1] (on écrit aussi `0xffff:0000` comme nous le verrons plus bas), couverte par la ROM du Bios. Cette instruction est elle-même, ou est suivie d'un, saut vers le cœur du code du BIOS. Ce dernier initialise les composants essentiels de la machine, émet des *beeps* lorsqu'une erreur est détectée (phase appelée POST pour *Power-On Self Test*), puis exécute les "extensions" situées entre les adresses `0xc0000` et `0xeffff` [12], parmi lesquelles figurent l'initialisation de la carte video, des interfaces IDE, etc...

Vient ensuite la recherche des chargeurs d'OS. Pour cela, le BIOS parcourt les périphériques de stockage qu'il supporte (disques, CDROM, USB, réseau, disquette, ...) à la recherche d'un premier secteur sur le périphérique (*ie* bloc de données de 512 octets), qui se terminerait par les octets `0x55aa`. Lorsqu'il l'a trouvé, le BIOS le charge en mémoire, et saute à sa première instruction.

Dans le cas d'une disquette, ce premier secteur, le *secteur de boot*, s'occupe de copier, depuis la disquette vers la mémoire, ce qui correspond au noyau de l'OS (figure 6), puisque tout l'OS ne tient pas dans les 512 octets que compte le secteur de boot. Dans le cas

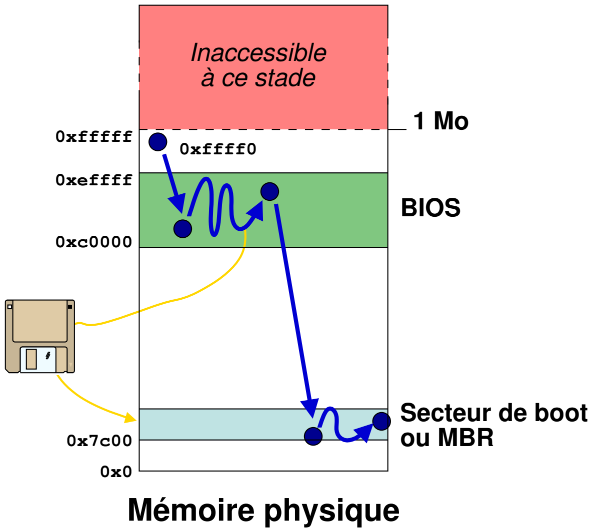


FIG. 6 – Premières étapes de démarrage d'un PC

de MSDOS par exemple, il parcourait les entrées du système de fichiers (FAT) à la recherche des entrées `IO.SYS` et `MSDOS.SYS`, et les chargeait en mémoire. Nous décrivons le cas Grub et lilo plus bas.

Dans le cas d'un disque dur, il y a une notion de *partition*, qui change très légèrement cette étape de chargement. Le premier secteur du disque est le "*MBR*" (*Master Boot Record*) bien connu de ceux qui ont déjà subi quelques déconvenues d'installation de système : qui n'a pas joué avec `fdisk /MBR`, ou `fixmbr`, ou `mbr` ? Une fois que le BIOS l'a chargé en mémoire et saute à sa première instruction, par défaut celui-ci s'occupe de charger en mémoire le premier secteur de la première partition du disque marquée "active" : le *partition loader*, puis il saute à la première instruction de ce secteur. Ce secteur aura ensuite la même fonction que le secteur de boot d'une disquette. Dans d'autres cas, le MBR ne s'occupe pas de charger le *partition loader*, mais prendra directement la fonction du secteur de boot de la disquette, même si plusieurs partitions sont définies sur le disque (Grub et lilo peuvent fonctionner ainsi).

4.2.2 Cas simple : SOS est chargé par Grub

Principes de Grub. Grub peut être installé sur une disquette, dans le MBR d'un disque dur, ou sur le premier secteur d'une partition d'un disque. Le premier secteur en question s'appelle `stage1` pour Grub, et il charge la deuxième partie du chargeur (`stage1.5` ou `stage2`), grâce à une liste de secteurs qui est stockée *en dur* à un endroit connu du disque déterminé au moment de l'installation du `stage1` (figure 7). Le rôle du `stage2` est de charger le menu de configuration, d'éventuellement initialiser la carte réseau, puis de charger l'OS sélectionné, car à ce niveau l'OS est simplement un ou plusieurs fichiers. Pour cela Grub est capable de connaître plusieurs formats de fichiers binaires (noyau Linux, format ELF, fichiers compressés, ...), plusieurs types de systèmes de fichiers (ext2/3, FAT, FFS), et même de piloter des cartes réseau : c'est une sorte de mini système d'exploitation, qui fait rigoler les amateurs de machines MacIntosch ou Sparc, puisque celles-ci proposent ces services en standard dans leur équivalent du BIOS.

Pour information, Lilo fonctionne selon le même principe que Grub, à la différence qu'il n'a aucune notion de système de fichiers, et ne connaît qu'un seul format de fichier binaire (noyau Linux). Il est seulement capable de stocker les listes de secteurs qui contiennent les différents OS à charger. C'est pourquoi il ne faut pas oublier de taper `lilo` avant de rebooter quand on a changé ou rajouté un noyau. On retrouve le même principe avec le chargeur de linux sur ppc (`yaboot`).

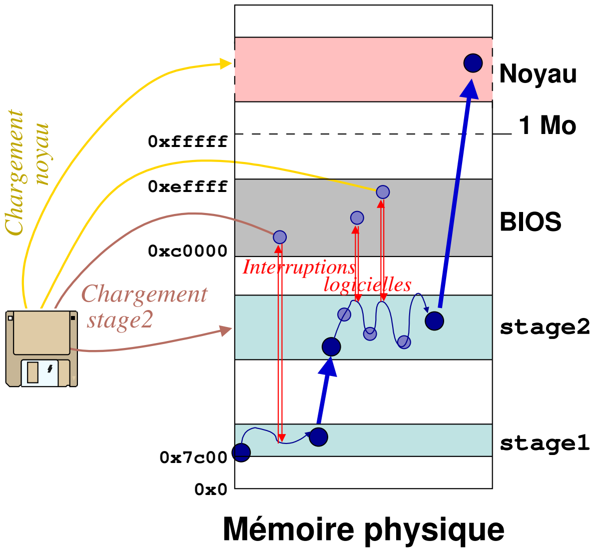


FIG. 7 – Etapes d'un chargeur de noyau

Pour effectuer les lectures depuis le périphérique par lequel ils ont été chargés, MSDOS, Grub, lilo, etc... font appel à une sorte de bibliothèque de fonctions "standards" fournie par le BIOS, et toujours accessibles suivant un mode lui aussi standard : les *interruptions logicielles*. Ils concernent Grub et lilo à ce niveau, nous ne les détaillons pas ici, mais nous serons amenés à le faire quand nous décrirons le secteur de boot que nous fournissons, en section 4.2.3.

Chargement de SOS par Grub. En ce qui concerne SOS, Grub charge le noyau, qui est un simple fichier, à une adresse en mémoire physique qui est spécifiée dans le binaire chargé. Cette information est fournie sous la forme d'une structure de données dont le format est spécifié par le standard multiboot [4], et qui doit se trouver dans les 8192 premiers octets du fichier à charger. Pour identifier cette structure, Grub balaye le début du fichier contenant le noyau à la recherche de marqueurs (un entier fixé par le standard, et nommé "*magic*").

Cette structure est `bootstrap/multiboot.h:struct multiboot_header`. Dans SOS, une telle structure est instanciée dans la section `.multiboot` du noyau (voir la section A.1), au tout début du fichier assembleur `bootstrap/multiboot.S`, et est très fortement inspirée des exemples accompagnant les sources de Grub :

```
/* The multiboot header itself. It must come first. */
.section ".multiboot"
/* Multiboot header must be aligned on a 4-byte boundary */
.align 4
multiboot_header:
  /* magic=        */ .long MULTIBOOT_HEADER_MAGIC
  /* flags=        */ .long MULTIBOOT_HEADER_FLAGS
  /* checksum=     */ .long -(MULTIBOOT_HEADER_MAGIC \
                            +MULTIBOOT_HEADER_FLAGS)
  /* header_addr=  */ .long multiboot_header
  /* load_addr=    */ .long __b_kernel
  /* load_end_addr=*/ .long __e_load
  /* bss_end_addr= */ .long __e_kernel
  /* entry_addr=   */ .long multiboot_entry
```

Entre autres choses, cette structure renferme :

`magic` : le marqueur de repérage (`0x1BADB002`),

`checksum` : une donnée de contrôle pour s'assurer que la structure en question a de fortes chances d'être effectivement du type `struct multiboot_header`,

`load_addr`, `load_end_addr` : les adresses de début et de fin de l'OS à charger en mémoire,

`entry_addr` : l'adresse de la première instruction de l'OS à exécuter une fois que l'OS sera chargé en mémoire.

Les valeurs inscrites dans ces champs sont en fait calculées soit directement par l'assembleur, soit lors de la phase d'édition de liens, comme nous le verrons en annexe A.

Lancement de SOS. Une fois le noyau chargé, Grub saute à l'adresse `bootstrap/multiboot.S:multiboot_entry` comme indiqué par le champ `entry_addr` du `multiboot_header`. Cette fonction effectue les opérations suivantes :

1. Changer de pile :

```
/* Set up a stack */
movl $(stack + MULTIBOOT_STACK_SIZE), %ebp
movl %ebp, %esp
```

   Nous préférons en effet ne pas reposer sur la pile initiale fournie par Grub, en en choisissant une autre dont nous maîtrisons l'emplacement, afin d'éviter tout risque d'écrasement. Nous utilisons pour cela la zone de taille `MULTIBOOT_STACK_SIZE` (16 ko, voir `bootstrap/multiboot.h`) définie par la variable `bootstrap/multiboot.S:stack`, donc directement allouée dans le binaire formant le noyau.

2. Appeler la fonction C `sos_main` définie dans un autre fichier, en lui passant 2 arguments : le contenu des registres `eax` et `ebx` :

```
/* Push the magic and the address on the stack, so that
   they will be the parameters of the cmain function */
pushl %ebx
pushl %eax

/* Call the cmain function (os.c) */
call EXT_C(sos_main)
```

   Le contenu de ces registres est donné par la norme multiboot. Le premier correspond au *magic* `0x2BADB002` et permet au noyau de vérifier qu'il a été chargé par Grub. Le deuxième contient l'adresse d'une structure de type `bootstrap/multiboot.h:multiboot_info_t` fournie par Grub. Cette structure permet de récupérer des informations sur la configuration de la machine (taille de la mémoire, emplacement de certaines zones réservées).

Le reste de l'exécution de l'OS reprend alors au début de la fonction C `sos/main.c:sos_main()` : voir la section 4.3.

4.2.3 Cas folklorique : chargement par le secteur de boot fourni

Le chargement par Grub est totalement satisfaisant, d'autant plus qu'il permet de démarrer SOS par énormément de moyens, tels que *via* le réseau. Cependant, il nous paraît intéressant de donner ici quelques indications sur la réalisation d'un secteur de boot, puisque ce sera l'occasion de présenter une partie du passé historique qui entoure le processeur `i80x86`. Cela nous permettra aussi d'introduire certains noms connus mais obscures pour le béotien ("mode réel", etc...). Cette section n'est cependant pas fondamentale pour la suite : considérez-la davantage comme une parenthèse "culture générale", que comme un passage à maîtriser dans les détails.

Le secteur de boot que nous proposons correspond au fichier `extra/bootsect.S`, et relève le triple défi 1/ de faire moins de 512 octets pour mériter le nom de *secteur* de boot, 2/ de charger un noyau depuis le lecteur de disquettes sans s'attarder sur les détails matériels du dialogue avec le contrôleur de disquette, 3/ de placer le noyau au-delà de la barre des 1Mo en mémoire physique (nous en reparlons plus bas). Nous le détaillerons assez peu ici, préférant nous attarder sur les caractéristiques de l'environnement dans lequel il s'exécute ; il est cependant suffisamment commenté pour être bien compréhensible.

Ce secteur de boot permet de remplacer Grub partiellement. "Partiellement" parce que : *i)* il ne permet de charger SOS que depuis un lecteur de disquettes, *ii)* il ne permet pas à SOS de récupérer simplement certaines informations importantes, telles que la taille de la mémoire par exemple (Grub fournit cette information dans la structure de type `bootstrap/multiboot.h:multiboot_info_t` passée au noyau). C'est d'ailleurs pour cette raison que **ce secteur de boot ne sera plus utilisable avec SOS à partir de l'article 3**. Rien ne vous empêche de le modifier pour récupérer la taille de la mémoire par balayage manuel, ou par interrogation du BIOS : c'est la seule fonctionnalité qui lui manque pour être utilisé au-delà de l'article 2. Sachez cependant qu'une telle détermination qui fonctionne sur toute machine est largement plus compliquée à réaliser qu'il n'y paraît.

État du processeur à son initialisation

Lorsque le secteur de boot a été chargé en mémoire par le BIOS, nous rajeunissons de plus de 20 ans puisque le processeur fonctionne dans le mode dit "réel". Ce "mode" correspond exactement au fonctionnement du processeur Intel `8086` de 1978.

Dans ce mode, les registres du processeur sont sur 16 bits, et le bus d'adresses est sur 20 bits (*ie* 1Mo) [13, section 9.1]. Le jeu d'instructions dans ce mode est dit "16 bits" car elles agissent sur des données et des registres de 16 bits, et non pas parce qu'elles seraient codées sur 16 bits (les instructions Intel sont de taille variable, souvent plus longue que 16 bits).

Adressage mémoire en mode réel [13, chapitre 16.1]. Puisque la mémoire est accédée par un bus sur 20 bits (*ie* 1 Mo) alors que les registres sont sur 16 bits, on utilise 2 registres pour les accès mémoire : un registre "général" ou spécialisé (pointeur d'instruction, de pile) dit d'"'offset", plus un des registres spéciaux dits "descripteur de segment" qui permettent de récupérer les 4 bits qui manquent pour passer de 16 à 20 bits. A partir du contenu de ces 2 registres, l'adresse mémoire est bêtement déterminée par l'arithmétique suivante : `RAM_addr = segment*16 + offset` (voir la figure 8). Tout cela est bien dommage quand on réalise qu'on utilise 2 registres sur 16 bits, non pas pour adresser 32 bits de mémoire, mais seulement 20 bits...

Il en découle que dans ce mode, on est en pratique souvent limité par des zones contiguës (*segments*) adressables de 64 ko de long (*ie* 16 bits), commençant à des adresses multiples 16 octets. D'où la fameuse barrière des 64ko des vieux programmes DOS, qui n'était pas infranchissable, mais qui était rédhibitoire puisqu'il fallait modifier à la main la valeur des registres de descripteurs de segment pour la dépasser... jusqu'à ce que le mode dit "protégé" du processeur apparaisse. Nous y reviendrons plus bas, dans la mesure où nous souhaitons que le secteur de boot charge SOS à partir de l'adresse 2 Mo en mémoire physique.

Plus amusant : puisqu'il y a une liaison déterministe entre les adresses manipulées par le processeur et les adresses mémoire, on a tendance à écrire les adresses mémoire sous la forme `segment:offset`, telle que "`0xffff:0000`" que nous avons vue plus haut. Or on peut remarquer qu'il y a plusieurs façons

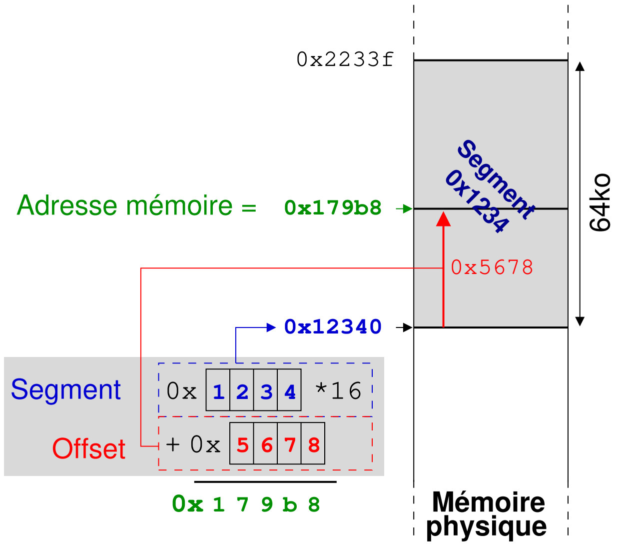


FIG. 8 – Adressage mémoire en mode réel, exemple avec l'adresse `0x1234 :5678`

de représenter chaque adresse physique avec la notation `segment:offset` précédente, puisqu'elle peut aussi s'écrire `{segment - x}:{offset + 16*x}` pour des valeurs de x raisonnables...

Accès au BIOS. Malgré les limitations de ce mode, son intérêt est que c'est quasiment le seul mode du processeur dans lequel le fonctionnement correct du BIOS est garanti. En effet, le BIOS est constitué d'instructions 16 bits, et ne fonctionnera donc que dans les modes 16 bits du processeur. Ainsi, dans le mode protégé dont nous reparlons plus bas, et qui possède un jeu d'instructions dit "32 bits", il n'est pas question de faire appel à ces fonctions du BIOS. Même s'il existe des extensions dites Bios32 pour ce mode, les non-compatibilités au "standard" suivant les cartes mères sont légions, et seul windows semble être capable de l'utiliser correctement. C'est pourquoi nous avons préféré effectuer les accès à la disquette par le BIOS 16 bits en mode réel.

Dans le mode réel donc, le premier point totalement standard concerne le moyen d'appeler les fonctions du BIOS : les *interruptions logicielles*. Le processeur s'attend en effet à trouver un tableau de 256 adresses (au format `segment:offset`, donc sur 4 octets) situé dans le premier kilo-octet de la mémoire physique : le *vecteur d'interruptions*. Dans ce vecteur, la i^{eme} adresse correspond à la première instruction de la routine de traitement de l'interruption *i*. Lorsque le secteur de boot fait appel à une interruption logicielle, par une instruction assembleur du type `int $0x10`, cela correspond schématiquement à un appel de la routine associée, avec certaines garanties sur les registres modifiés. L'intérêt de cette méthode est que l'adresse précise de la routine appelée n'a pas à être connue par le secteur de boot, qui pourra alors y faire appel, quelle que soit la façon dont les routines sont stockées dans le Bios : seul le numéro d'interruption logicielle doit être connu.

Le terme "interruption logicielle" est bien sûr à rapprocher de "interruption matérielle" (voir la section 2.3). En fait, dans le mode réel, les 16 premières entrées du vecteur d'interruptions correspondent directement aux lignes d'interruptions matérielles, et le reste est réservé aux interruptions logicielles. En mode protégé, le principe général est le même bien que la configuration soit légèrement différente, nous en reparlerons dans le prochain article.

La deuxième caractéristique standard des Bios accédés dans le mode réel, est que ceux-ci doivent prendre en charge certaines entrées du vecteur d'interruption, et les routines associées doivent avoir un comportement *relativement* normalisé [14, 15] (ladite norme étant majoritairement *de fait*, elle regorge d'incompatibilités, suivant les fabriquants de cartes mères). Parmi elles figurent les interruptions logicielles `0x10` (affichage à l'écran) et `0x13` (contrôleur de disquette) que nous utilisons dans notre secteur de boot.

Lorsque une interruption logicielle BIOS est levée, le BIOS s'attend à trouver l'identification de la *fonction* demandée : lecture d'un secteur de la disquette, déplacement de la tête de lecture, démarrage du moteur, ... Cette fonction est indiquée dans les 8 bits de poids fort du registre `ax` (`ah`), et il peut éventuellement être complété d'identifiants de sous-fonctions (`al`, les 8 bits de poids faible de `ax`), et de paramètres (autres registres).

Copie du noyau en mémoire

Revenons à notre secteur de boot `extra/bootsect.S`. Sa première partie fonctionne en mode réel, et s'occupe de copier le noyau depuis la disquette vers les adresses "basses" de la mémoire, c'est-à-dire situées en deçà de 1Mo.

A l'issue du boot, le BIOS place systématiquement le secteur de boot à l'adresse `0x7c0:0000` en mémoire physique (figure 9(a)). Le BIOS l'identifie en tant que secteur de boot grâce aux deux octets `55aa` placés en toute fin de 512 octets (fichier `extra/sos_bsect.lds`). Le secteur de boot commence (label `_bsect` puis `here`) par se déplacer à une adresse plus élevée (`0x9F00:0000`) afin de libérer le maximum de mémoire physique contigüe en deçà des 1Mo (figure 9(b)) :

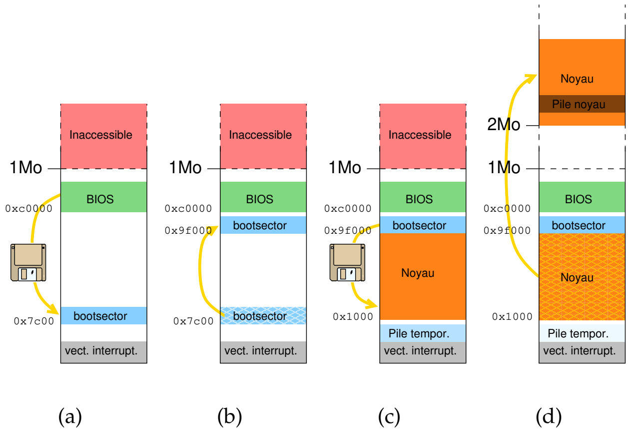


FIG. 9 – Etapes de chargement du noyau par le secteur de boot

```
movw $(BOOT_SIZE>>1), %cx
movw $COPY_SEG, %ax
movw %ax, %es
cld
rep ; movsw
```

Ensuite, grâce à l'interruption logicielle `0x13`, le secteur de boot charge les `load_size` secteurs formant le noyau, un par un depuis la disquette vers la mémoire physique, à partir de l'adresse 4ko (figure 9(c)). Le symbole `load_size` est défini par un script `ld` (voir la section A.2). Le cœur de la boucle qui charge les secteurs prend la forme suivante :

```
pushw %es
movw $0x0201, %ax /* service 0x2, chargement 0x1 secteur */
int $0x13          /* Go ! */
jc halt            /* erreur */
popw %ax
addw $32, %ax      /* on a charge un secteur, donc on doit
                    * charger 512 bytes plus loin . Pour cela,*/
movw %ax, %es      /* on avance donc le segment du buffer de
                    * 32bytes, ie 1 secteur en RAM (car 32*16=512) */
decw (load_size)   /* et on repart pour le prochain secteur
                      tant qu'on n'a pas fini ! */
jnz .nextsector    /* Iteration suivante */
```

Compte tenu de la méthode utilisée (incrémentation du segment plutôt que de l'offset), la taille à charger n'est pas limitée à 64Ko. Cette taille est en fait limitée à la taille de la mémoire disponible entre l'adresse de chargement du noyau (`0x1000`), et l'adresse de début du secteur de boot (`0x9f000`), soit environ 640 ko.

Passage en mode protégé et déplacement du noyau à sa destination

Nous avons "profité" du mode réel pour pouvoir utiliser le BIOS, et ainsi ne pas avoir à utiliser de driver spécialisé afin de piloter le contrôleur de disquette. Nous devons maintenant trouver un moyen de positionner le noyau chargé à partir de `0x1000`, vers son adresse définitive (`0x200000`, *ie* 2 Mo), puisque cette adresse est au-delà de la barrière fixée par le bus d'adresses (sur 20 bits, soit 1 Mo) du mode réel (figure 9(d)).

Pour cela, nous allons passer le processeur dans le mode dit *protégé* [13, chapitres 2 et 3], le mode dans lequel le processeur fonctionne au sortir de Grub, et le mode dans lequel fonctionne le reste de SOS. Dans ce mode, les registres, les données et les adresses sont sur 32 bits, ce qui permet entre autres d'accéder aux adresses au-delà de 1Mo. Ce mode n'est accessible que sur les processeurs `80386` et suivants, bien qu'il soit en partie disponible dès le `80286`. Comme nous l'avons fait remarquer (section 4.2.3), nous considérons qu'une fois en mode protégé, il serait déraisonnable d'appeler les fonctions du bios/bios32.

Le secteur de boot passe donc en mode protégé (label `GoPmode`), configuré de telle sorte que l'ensemble de l'espace mémoire physique 0-4Go est accessible directement : il s'agit de configurer la segmentation (label `InitGDT`) dans le mode le plus simple, dit *flat mode* dont nous aurons l'occasion de reparler dans le prochain article, et de changer le mode en modifiant le bit 0 du registre dit "de contrôle" `cr0` [13, section 9.8] :

```
  /* Préparation du flat mode */
InitGDT:
  lgdt gdtr /* gdtr est définie en fin de fichier */

  /* Passage en mode protégé */
GoPMode:
  movl %cr0, %eax
  orb  $1, %al /* set PE bit to 1 */
  movl %eax, %cr0
```

La dernière opération à effectuer (label `MoveKernelToFinalAddr`) consiste enfin à copier le noyau de son ancienne adresse (`0x1000`) vers la nouvelle :

```
cld
/* On commence la copie au debut du noyau : */
movl $LOAD_ADRESS, %esi
/* On copie vers cette adresse : */
movl $FINAL_ADDRESS, %edi
/* Taille recopie : */
movl $MAX_KERN_LEN, %ecx
shrl $2, %ecx
rep movsl
```

Puis (label `LaunchKernel`) le processeur saute à l'adresse du symbole `sos_main` après avoir changé de pile (comme `bootstrap/multiboot.S` le faisait, voir la section 4.2.2).

4.3 Cœur de SOS

Que ce soit après un chargement par Grub, ou par le secteur de boot fourni, au moment où la première instruction de `sos_main()` est exécutée :

- Le processeur est en mode protégé, l'intégralité de la mémoire est accessible directement, la pagination n'est pas activée (nous en reparlerons dans l'article 3),
- La mémoire physique est supposée occupée par 3 blocs : 1 bloc pour le noyau (symboles `__b_kernel` - `__e_kernel`, aux alentours de 2Mo), et 2 blocs pour le BIOS (nous y reviendrons dans les articles suivants). Tout le reste est totalement disponible,
- La pile se trouve dans une zone de 16ko dans l'espace mémoire occupé par le noyau,

– Si SOS a été chargé par Grub, les deux paramètres `magic` et `addr` de `sos/main.c:sos_main()` sont positionnés respectivement à `MULTIBOOT_BOOTLOADER_MAGIC` et l'adresse d'une structure de type `multiboot_info_t` remplie par Grub. Sinon, ces deux paramètres sont indéfinis.

Après avoir initialisé les deux "pilotes" `bochs` et `x86_videomem`, `sos/main.c:sos_main()` affiche un message à l'écran dont le contenu dépend de la façon dont SOS a été chargé (Grub ou secteur de boot). Puis il envoie un message sur la sortie de debuggage de `bochs/qemu` si tous deux ont été compilés avec cette fonctionnalité activée :

```
/* Greetings from SOS */
if (magic == MULTIBOOT_BOOTLOADER_MAGIC)
  /* Loaded with Grub */
  sos_x86_videomem_printf(1, 0,
                         SOS_X86_VIDEO_FG_YELLOW
                           | SOS_X86_VIDEO_BG_BLUE,
                         "Hello Grub");
else
  /* Not loaded with grub */
  sos_x86_videomem_printf(1, 0,
                         SOS_X86_VIDEO_FG_YELLOW
                           | SOS_X86_VIDEO_BG_BLUE,
                         "Hello SOS boot sector");

sos_bochs_putstring("Message in a bochs\n");
```

Ces quelques étapes permettent de tester le chargement, l'affichage, et les fonctions de type `printf()`.

Enfin, SOS entre dans une boule infinie, qui risque de faire rebooter certains processeurs overclockés (dans ce cas, remplacer `continue;` par `asm("hlt;");`). Dans un vrai OS, cette boucle s'appelle souvent la "tâche *idle*" ; en attendant d'avoir du multitâche, nous n'avons pas d'autre choix que cette solution si nous ne voulons pas faire rebooter la machine immédiatement.

Nous décrivons ci-dessous les 2 pilotes utilisés. Mais avant, décrivons la bibliothèque de fonctions C, quelques fichiers d'en-tête, quelques types et macros importantes.

4.3.1 Bibliothèque C

Dans un noyau, on ne dispose que des fonctions qu'on a codées, qui doivent être intégrées directement dans le noyau. Or il est des fonctions classiques de la bibliothèque C habituelle dont on aimerait profiter, comme `memcpy`, `strcmp` par exemple. Nous définissons donc une mini-bibliothèque C qui renferme pour l'essentiel les fonctions qu'on trouve sous Unix dans `string.h`. Ces fonctions sont déclarées dans `sos/klibc.h` et sont définies dans `sos/klibc.c`.

On pourra remarquer que les fonctions telles que `strcpy` et `strcat` ne sont pas définies. Nous considérons en effet que ces fonctions sont très dangereuses à utiliser : par inadvertance, il est très facile de provoquer un débordement de buffer, dans le cas où les chaînes destination sont trop courtes pour contenir le résultat de l'opération. C'est pourquoi nous imposons d'utiliser leurs équivalents plus robustes que sont respectivement `strzcpy` et `strzcat`. Ces fonctions sont l'analogue des classiques `strncpy` et `strncat`, elles-mêmes plus sûres que `strcpy` et `strcat`, mais sont encore plus robuste puisqu'elles introduisent systématiquement un zéro pour terminer la chaîne résultat.

Ces fichiers définissent également une fonction qui se situe habituellement dans `stdio.h` : `vsnprintf(char *dest, sos_size_t max_size, const char *fmt, va_list ap)`. Cette fonction remplit une chaîne résultat de taille maximale `max_size` donnée (d'où le `n` dans le nom) en fonction du *format* `fmt` et des paramètres indiqués dans le tableau `ap` que l'on accède séquentiellement grâce à la macro `va_arg` :

```
int vsnprintf(char *buff, sos_size_t len,
              const char * format, va_list ap)
{
 ...
  ...
  /* Traitement des clefs '%d' : */
  case 'd':
  {
    /* Recuperation du parametre associe */
    int integer = va_arg(ap,int);

    /* Affichage de integer */
    ...
  }
  break;
  ...
 ...
}
```

De par la simplicité du code de cette fonction, le format est limité aux clefs `%s`, `%c`, `%d`, et `%x`, et sans les modificateurs (comme `%08x`).

Nous proposons également `snprintf`, la variante *variadique* de cette fonction, c'est-à-dire celle qui remplit la chaîne destination en fonction du format, et d'un nombre variable d'arguments. Cette fonction appelle `vsnprintf`, et utilise les macros standards `va_start` et `va_end` pour indiquer le tableau des paramètres `ap` de `vsnprintf` à partir des arguments en nombre variable de `snprintf`. Dans le cas de l'architecture `i80x86`, les macros `va_start`, `va_arg` et `va_end` sont simplement une façon de calculer des adresses dans la pile (voir la figure 10), puisque celle-ci contient tous les paramètres des fonctions les uns à la suite des autres (sauf en cas d'optimisation explicitement spécifiée par le programmeur : mot-clef "`register`") : le tableau `ap` est simplement un morceau de la pile, et ces macros permettent d'y accéder séquentiellement.

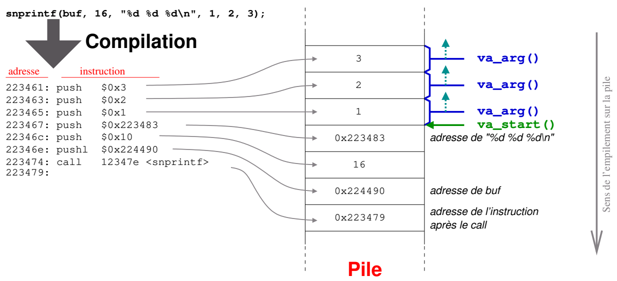


FIG. 10 – Principe des macros `va_start` et `va_arg` : état de la pile au début d'un appel à `snprintf()`

4.3.2 Types et macros importants

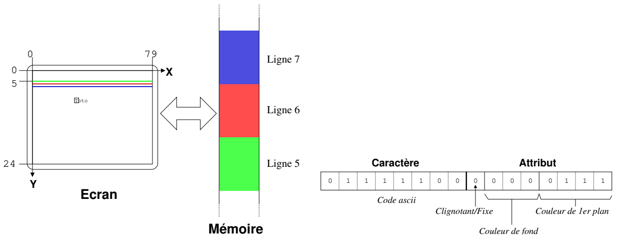
Fichier `sos/types.h`**.** Dans un OS, étant donné qu'on interagit régulièrement avec le matériel, on a souvent besoin de manipuler des données ou des éléments de structures dont on doit parfaitement maîtriser la taille en nombre d'octets. Afin que ces manipulations soient explicitement indiquées dans le code, on définit des alias du genre `sos_ui32_t` (entier sur 32 bits non signé), etc... C'est dans ce fichier qu'on définit également d'autres alias destinés à faciliter la compréhension du code, comme par exemple `sos_size_t` pour désigner une taille en octets, ou `sos_count_t` pour désigner un nombre d'objets, etc... C'est dans ce fichier que nous définissons également quelques macros fortement associées à des types, comme `TRUE` et `FALSE` associées au type `sos_bool_t`, et `NULL` associée au type pointeur générique (`void*`). Ce fichier sera enrichi par la suite.

Fichier `sos/errno.h`**.** Dans tout logiciel, on ressent le besoin de pouvoir indiquer qu'une fonction se termine correctement, ou qu'une erreur a été détectée. Dans ce cas, on souhaite avoir une petite idée de la cause de l'erreur : arguments invalides, état incorrect, fonction pas encore implantée... Dans SOS, ce fichier définit le type `sos_ret_t` destiné à être le type de la valeur de retour de la plupart des fonctions. Le principe est habituel : lorsqu'une fonction renvoie une valeur nulle (ou `SOS_OK`) ou positive, c'est qu'elle s'est déroulée correctement. Sinon c'est qu'une erreur s'est produite, et la cause de l'erreur correspond alors à l'opposé d'une des macros définies dans ce même fichier. Par exemple, pour signaler que les paramètres de la fonction sont invalides, il suffit d'écrire : `return -(SOS_EINVAL);`.

Fichier `sos/assert.h`**.** Dans tout logiciel, on ressent le besoin de pouvoir signaler immédiatement à l'utilisateur que quelque chose d'anormal se produit. Cela se fait souvent par des *assertions*, c'est-à-dire des fonctions qui vérifient que la condition fournie en paramètre est vraie, et qui affichent un message si elle s'avérait fausse. Il y a plusieurs manières de réagir dans le cas où la condition est fausse : soit on sort automatiquement de la fonction courante avec un résultat d'erreur (`return`), soit on termine le programme. Dans le cas de SOS, nous n'avons retenu pour le moment que la deuxième solution, prise en charge par la macro `SOS_ASSERT_FATAL(condition)`. Lorsque la condition est fausse, un message est affiché à l'écran (et sur la console si bochs et qemu ont été compilés avec le port `0xe9` supporté) qui affiche une ligne au format suivant, rappelant la condition qui est fausse :

```
sos_main@sos/main.c:59 Assertion TRUE == FALSE failed
```

Puis la machine entre dans une boucle infinie : le système est ainsi bloqué en cas d'échec, il attend d'être rebooté.



(a) Lien écran ↔ mémoire video

(b) Détail d'un élément

FIG. 11 – Mémoire video en mode caractères

4.3.3 Pilote `x86_videomem`.

Sur l'architecture PC originelle, une zone d'adresses physiques correspondant à la "mémoire vidéo" est *mappée* en mémoire physique (voir la figure 11). Il s'agit d'une version basique, et en mode texte seulement pour l'instant, des *framebuffers* classiques sur d'autres architectures, et qui sont également redéfinis par Linux pour l'architecture PC.

Suivant que la carte vidéo est monochrome ou couleurs, l'adresse de base de ce *mapping* video varie [14][16, partie DOS chapitre 23]. Nous avons supposé que SOS allait fonctionner sur des PC avec une carte capable de gérer la couleur, même si l'écran est noir et blanc. Dans ces conditions, la zone commence à l'adresse physique `0xB8000` par défaut, et "mesure" $80 * 25 * 2$ octets. Cette zone correspond à un tableau à 2 dimensions (figure 11(a)) : 25 lignes de 80 colonnes. Chaque élément correspond à un caractère affiché à l'écran, et est codé sur 2 octets : le premier correspond au code ASCII du caractère, et le deuxième aux *attributs* du caractère. Ce dernier correspond à la couleur du caractère, à la couleur de l'arrière-plan du caractère, et peut indiquer un soulignement, un clignotement, etc... La figure 11(b) donne sa structure ; nous laissons la détermination des codes de couleurs à l'expérimentation du lecteur curieux.

Dans `drivers/x86_videomem.c` cette zone est déclarée directement comme étant un tableau C à deux dimensions, dont chaque élément est une structure C à deux champs correspondant directement à la définition ci-dessus (nous reparlerons de l'attribut "`packed`" dans le prochain article) :

```
/** The structure of a character element in the video memory. @see
    http://webster.cs.ucr.edu/AoA DOS edition chapter 23 */
typedef struct {
  unsigned char character;
  unsigned char attribute;
} __attribute__ ((packed)) x86_video_mem[LINES*COLUMNS];
```

Ainsi, les fonctions définies dans ce fichier et déclarées dans `drivers/x86_videomem.h` ne font qu'utiliser ce tableau directement. Il est possible d'effacer très simplement l'écran (`sos_x86_videomem_cls()`), d'afficher très simplement des chaînes de caractères à un emplacement donné avec un attribut donné (`sos_x86_videomem_putstring()`):

```
sos_ret_t sos_x86_videomem_putstring(unsigned char row,
                                     unsigned char col,
                                     unsigned char attribute,
                                     const char *str)
{
  unsigned video_offs = row*COLUMNS + col;

  for ( ;
       str && *str && (video_offs < LINES*COLUMNS) ;
       str++, video_offs++)
    {
      (*video)[video_offs].character = (unsigned char)*str;
      (*video)[video_offs].attribute = attribute;
    }

  return SOS_OK;
}
```

Une fonction combinant à la fois la précédente avec `vsnprintf()` (voir la section 4.3.1) est également définie.

Il est important de noter que ce "pilote" est extrêmement basique. Il n'interprète en effet aucun caractère spécial, tel que `\n` ou `\r` ; il faut donc passer à la ligne, ou revenir en début de ligne, à la main. En corollaire, il ne gère pas le *scrolling*, c'est-à-dire le fait de faire défiler le contenu de l'écran vers le haut quand on essaye d'écrire au delà du bas de l'écran. Bien sûr, dans un prochain article nous implanterons ces fonctionnalités dans un vrai driver de plus haut niveau, qui utilisera les fonctions de celui-ci.

4.3.4 Pilote `bochs`.

Le "pilote" pour bochs est encore plus basique que le précédent, puisqu'il n'y a pas de notion de position, ni d'attribut : chaque caractère envoyé à ce pilote est envoyé directement sur le terminal où bochs/qemu a été lancé. En conséquence, contrairement au pilote précédent, les caractères `\n` ou `\r` sont correctement interprétés, puisque c'est le terminal qui s'en charge. Bien entendu, sur machine réelle, on ne peut récupérer aucun de ces caractères, car le port `0xe9` sur un PC n'est en général pris en charge par aucun matériel précis.

Ce "pilote" repose sur les fonctions définies dans `drivers/bochs.c` et déclarées dans `drivers/bochs.h`, très simples et analogues à celles du pilote précédent. Ces fonctions utilisent le fichier `hwcore/ioports.h` qui définit les macros C permettant de générer les instructions machine pour les accès aux ports d'entrée/sortie (voir la section 2.2). C'est grâce à ces macros qu'on peut envoyer chaque caractère au fameux "port `0xe9`" de bochs/qemu, qui s'occupe de le retransmettre sur le terminal où bochs/qemu a été lancé :

```
sos_ret_t sos_bochs_putstring(const char* str)
{
  for ( ; str && (*str != '\0') ; str++)
    outb(*str, 0xe9)

  return SOS_OK;
}
```

4.3.5 Quelques remarques sur les conventions de programmation

Pour terminer ce tour d'horizon du code, voici quelques règles que nous nous sommes fixées. Elles sont criticables, mais nous nous y conformerons pour une plus grande homogénéité du code :

- Les commentaires sont de type *doxygen*,
- Les commentaires sont en anglais, de même que les identifiants. Sauf pour le secteur de boot,
- Les identifiants de types, de fonctions et de variables globales sont précédés par "`sos_`",
- Nous n'utilisons pas de `typedef` pour les définitions de structures, afin de rendre explicites les variables et paramètres qui sont d'un type composé. Sauf pour certains cas très particuliers (`atomic_t` dont nous parlerons dans un prochain article),
- Toutes les fonctions "exportées" (*ie* sans le mot-clef `static`) doivent renvoyer une valeur : soit un pointeur ou une adresse qui vaut `NULL` en cas d'erreur, soit un élément de type `sos_ret_t` (voir la section 4.3.2),
- Nous employons le mot-clef `const` dans les paramètres de fonctions si possible.

4.4 Testons

4.4.1 Machines PC avec Grub installé : simulation ou test grandeur nature

Pour les heureux possesseurs de PC sur lesquels Grub est installé, ils peuvent compiler et fabriquer l'image avec leur Grub, puis tester sur un vrai PC ou dans un simulateur (bochs ou qemu).

Compilation. Taper la commande "`make`". Cela fabriquera le noyau `sos.elf` (au format ELF pour chargement par Grub et/ou debuggage avec `gdb`), l'image de disquette `fd.img`, et aussi le fichier `sos.map` qui rassemble les correspondances $adresse \leftrightarrow symbole$. La commande `make clean` effacera les fichiers générés pour repartir sur des bases saines pour une future compilation. **Attention :** les dépendances ne sont pas gérées, ce qui veut dire que si vous modifiez un fichier d'en-tête, il faut faire `make clean` avant de recompiler.

Simulation dans bochs. Modifier votre fichier de configuration de bochs : `~/.bochsrc`, pour indiquer où se trouve l'image de la disquette sur laquelle il doit booter. Exemple de fichier de configuration :

```
floppya:     1_44=/home/d2/bochs/fd.img, status=inserted
romimage:    file=/usr/share/bochs/BIOS-bochs-latest, address=0xf0000
vgaromimage: /usr/share/vgabios/vgabios.bin
megs:63 # 63 Mo de RAM
```

Puis ensuite, il suffit de lancer la commande `bochs` pour voir s'afficher la fenêtre avec le message de bienvenue de SOS. Si bochs a été compilé avec `--enable-port-e9-hack` (voir la section 3.3), un autre message de bienvenue devrait aussi apparaître dans le terminal où bochs a été lancé.

Si bochs a été compilé avec `--enable-debugger` et `--enable-disasm`, il offre un *prompt* qui permet d'office de bénéficier de quelques commandes classiques de `gdb` (pas à pas, breakpoints, ...), ce qui est pratique pour le debuggage. Si on utilise en parallèle la sortie de `objdump -S sos.elf`, ou la commande `load-symbols "sos.map"` du debugger de bochs, il est possible de faire le lien entre les instructions exécutées et le code source.

Simulation avec qemu. Il suffit de lancer la commande `qemu -fda <chemin_vers_fd.img>`, puis la fenêtre du simulateur s'affiche avec le message de bienvenue de SOS. Pour voir l'autre message de bienvenue sur le terminal où qemu a été lancé (équivalent au `--enable-port-e9-hack` de bochs), il faut avoir compilé qemu après avoir adapté et/ou appliqué le patch `extra/qemu-port-e9.diff`. Si qemu reste bloqué dans Grub, installez une version récente (`cvs`) de Grub et/ou de qemu.

On peut aussi debugger le noyau par `gdb` de la même façon qu'un programme normal : il suffit de lancer `qemu -s -fda <chemin_vers_fd.img>` dans un terminal, et de lancer `gdb sos.elf` dans un autre, puis de taper `target remote localhost:1234` au prompt de `gdb`. Il est possible de faire la même chose avec bochs, à condition de l'avoir compilé avec l'option `--enable-gdb-stub`.

Sur machine réelle. Il y a deux options. Soit Grub est déjà installé sur la machine, auquel cas il suffit de modifier son fichier de configuration pour lui dire de charger le noyau `sos.elf` (directive "`kernel=`"); le noyau peut éventuellement être sur une autre machine si on a compilé Grub avec le support d'une carte réseau. Soit on teste en bootant sur une disquette sur laquelle on aura transféré l'image `fd.img` par une commande du type "`dd if=fd.img of=/dev/fd0u1440 bs=18k`". Une fois le message de bienvenue de SOS affiché, il faut éteindre/rallumer ou rebooter la machine par appui sur les boutons idoines, puisque la combinaison de touches `ctrl-alt-del` n'est pas gérée car SOS n'a pour l'instant aucun pilote pour le clavier.

4.4.2 Machines sans Grub

Si Grub n'est pas installé sur la machine sur laquelle on compile, SOS vient avec une image de disquette contenant une installation de Grub correcte : `extra/grub.img.gz`. La seule étape requise est donc le transfert du noyau sur cette image, et la confection du menu de Grub.

Plutôt que de modifier le `Makefile`, nous avons rajouté la possibilité de le personnaliser à l'aide d'un fichier externe. Ce fichier doit s'appeler "`.mkvars`" et doit être placé à la racine de SOS. Il surchargera les commandes par défaut du Makefile ; quelques avertissements sans gravité concernant cette surcharge s'afficheront d'ailleurs, mais on peut les ignorer.

Vous pourrez reprendre quasiment *in extenso* le fichier d'exemple `extra/dot.mkvars`, le copier en tant que `.mkvars` à la racine de SOS, et commenter les lignes `CC=` à `OBJCOPY=` qui servent à la cross-compilation de SOS (voir ci-dessous). Une fois cette manipulation effectuée, "`make`" générera l'image de la disquette `fd.img` en utilisant l'installation de Grub fournie avec les sources. Sur les vieilles versions des `mtools`, il se peut que la génération de l'image échoue : optez pour des `mtools` récents, ou enlevez `filter` dans `extra/mtoolsrc`.

4.4.3 Machines non-PC : simulation

Pour les machines qui ne sont pas des PC, il est tout de même possible de compiler et tester SOS. La compilation requiert un cross-compilateur (voir la section 3.1), et les mtools. Et le test fonctionne avec bochs et qemu, ou sur un PC disponible dans les parages.

Pour compiler, il faut suivre la même manipulation que pour les machines sans Grub précédemment, sauf qu'on conserve l'affectation des variables de `CC=` à `OBJCOPY=` du `.mkvars` : on les personnalise même pour les faire pointer vers les outils de cross-compilation corrects. Puis après `make` génère l'image de la disquette `fd.img` qu'il est possible de tester dans bochs ou sur une machine PC (en fabriquant une disquette, ou en demandant au Grub du PC de charger `sos.elf` via le réseau).

C'est la méthode principale que j'utilise, puisque SOS est développé sur un ibook : j'utilise bochs, ou un vieux PC en lui faisant charger le noyau par Grub depuis l'ibook via le réseau.

4.4.4 Articles 1 et 2 seulement : test avec le secteur de boot

Les méthodes de test précédentes reposaient toutes sur un chargement du noyau par Grub, même si Grub n'était pas disponible sur la machine de compilation. Pour fabriquer une image de disquette avec le secteur de boot qu'on fournit (voir la section 4.2.3), il faut commencer par compiler le noyau normalement (voir ci-dessus). Puis ensuite, faire : "`make -C extra`". L'image de la disquette est alors `extra/sos_bsect.img` pour test sur machine réelle (par copie sur vraie disquette) ou avec bochs. Elle s'appelle `extra/sos_qemu.img` pour tester avec qemu (voir `Makefile` qui explique pourquoi 2 images sont fabriquées).

Escale

Le moment est venu de faire une escale dans notre croisière. Nous espérons que cet article n'aura été ni trop prétentieux dans ses nombreux objectifs, ni trop vague pour tous les évoquer. Pour les détails plus techniques que nous avons omis de présenter ici, les sources devraient être largement commentées pour les appréhender.

Á vous maintenant de poursuivre l'aventure par des tests et expérimentations... en attendant le prochain article. Au programme de cet article : les interruptions matérielles et les exceptions. Bon vagabondage !

A Annexe : édition de liens et scripts `ld`

Lorsque nous écrivons : `int toto; void f(int a) { toto = a; }`, le compilateur génère un fichier *objet* du type `toto.o`, qui contient 2 *symboles* `f` et `toto`. Un symbole correspond à une adresse faisant référence à un élément du fichier objet ; dans notre exemple : une fonction `f` et un entier `toto`. Avant de pouvoir exécuter un programme, les symboles doivent être *résolus*, c'est-à-dire que les adresse doivent être déterminées, et inscrites dans le binaire final (figure 12). Nous donnons ici un aperçu des étapes qui mènent de la compilation au fichier exécutable final, et les éléments qui interviennent.

Les symboles peuvent être *internes* au fichier objet courant (par exemple la référence à `multiboot_entry` dans la structure `bootstrap/multiboot.S:multiboot_header`). Il peut aussi s'agir de références aux symboles *externes* au fichier courant (par exemple : `__e_load` référencé dans `bootstrap/multiboot.S`, mais qui n'y est pas défini). Dans tous les cas, c'est l'étape d'*édition de liens* (programme `ld` appelé manuellement, ou par `gcc -o ...`) qui s'occupe de la résolution de symboles.

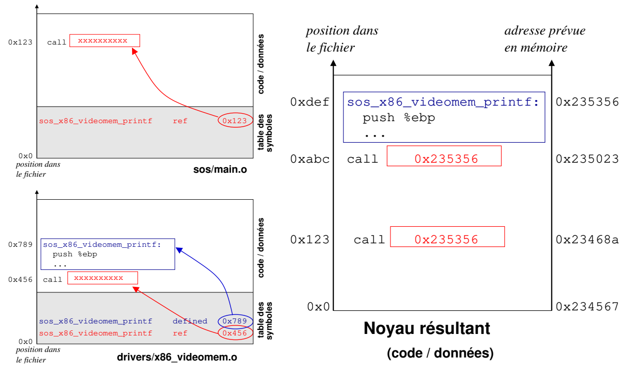
Pour cela l'éditeur de liens croise les symboles à résoudre dans chaque fichier objet (figure 12(a)), avec les symboles définis dans les autres fichiers objet spécifiés lors de l'édition de liens (figure 12(b)). Par défaut, tous les symboles définis dans n'importe lequel des fichiers objet peuvent être utilisés par l'édition de liens. Cependant, si on déclare un symbole avec le mot-clef `static` dans le source, on interdit qu'il soit utilisé par l'éditeur de liens, et il ne pourra donc pas être référencé par les autres fichiers objet.

Exemple de liste des symboles non résolus de `sos/main.o`, et leur emplacement dans le fichier :

```
d2@bingo article1 >readelf -r sos/main.o
Section de relocalisation « .rel.text » à l'adresse de décalage
0x554 contient 18 entrées:
 Décalage    Info     Type              Val.-sym    Noms-symboles
0000000d  00000802 R_386_PC32           00000000    sos_bochs_setup
00000012  00000902 R_386_PC32           00000000    sos_x86_videomem_setup
...
00000123  00000a02 R_386_PC32           00000000    sos_x86_videomem_printf
...
```

Exemple de liste des symboles définis dans `drivers/x86_videomem.o`, avec leur adresse dans le fichier, utilisée par l'éditeur de lien pour résoudre les symboles non résolus de `sos/main.o` :

```
d2@bingo article1 >readelf -s drivers/x86_videomem.o
Table de symboles « .symtab » contient 13 entrées:
   Num:    Valeur Tail Type    Lien   Vis      Ndx Nom
```



(a) Fichiers objet initiaux

(b) Après édition de liens et relocation

FIG. 12 – Edition de liens et relocation (schéma simplifié)

```
...
  9: 00000789   157 FUNC    GLOBAL DEFAULT    1 sos_x86_videomem_printf
...
```

En réalité, l'action de l'éditeur de liens est plus compliquée, car il doit aussi générer les adresses des symboles puisqu'elles ne sont pas définis lors de la confection des fichiers objet par l'assembleur : c'est l'étape de *relocation*, qui accompagne l'étape de résolution des symboles. Pour cela, il devra éventuellement respecter un "plan mémoire" fourni sous la forme d'un *script* `ld`. Dans le cas de SOS, nous avons deux tels fichiers : `support/sos.lds` et `extra/sos_bsect.lds`, suivant qu'on veut charger l'OS par Grub ou par le secteur de boot fourni.

Nous donnons ci-dessous quelques détails sur les deux types de fichiers manipulés par l'éditeur de liens `GNU ld` : les fichiers objet, et les scripts `ld`.

A.1 Structure d'un fichier objet

Les fichiers objet manipulés par les éditeurs de liens doivent au moins renfermer les informations sur la table des symboles définis ou appelés par le fichier, pour que la résolution de symboles soit possible. Il a donc été nécessaire de donner une structure standard à ces fichiers binaires, pour que l'éditeur de liens puisse situer précisément ces tables. Pour plus de souplesse, de généralité, et d'efficacité, on profite de ce besoin de structure pour découper le programme en régions logiques : les *sections*. Ces sections sont des morceaux contigus de chaque fichier objet, qui servent à rassembler le code, les données utiles au programme, à l'éditeur de liens, ou au debugger, et qui sont caractérisées par des contraintes (adresse de début, alignement, droit d'accès lecture/écriture/exécution). Sous Unix, les sections les plus courantes sont :

`.text` : le code machine,

`.data` : les données globales initialisées (par exemple `int globvar=0xdeadbeef;`),

`.rodata` : les données globales initialisées, et en "lecture seule". Typiquement : les chaînes de caractères (par exemple la chaîne `"bonjour\n"` dans `printf("bonjour\n");`),

`.bss` (qui signifie *Block Started by Symbol*, sens plutôt cryptique) : les données globales non initialisées (par exemple `int globvar;`).

Il existe plusieurs *formats* de fichiers binaires (`a.out`, `coff`, `elf`, ...); tous définissent la notion de section. Parmi eux, `elf` [17] est de loin le plus souple et permet de définir autant de sections qu'on le souhaite, avec les noms qu'on désire. D'ailleurs, la table des symboles est stockée dans une section (`.symtab`), la table des résolutions à effectuer également (sections `.rel.*`), et même la liste des noms de sections ! Pour avoir la liste des sections, voire même la liste des tables de symboles et de relocation, voir les outils `objdump` (tous formats binaires) et `readelf` (ELF seulement) :

```
d2@bingo article1 >readelf -S sos.elf
Il y a 9 en-têtes de section, débutant à l'adresse de décalage 0x2d48:

En-têtes de section:
  [Nr] Nom               Type            Adr      Décala.Taille ES Fan LN Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .multiboot        PROGBITS        00200000 001000 000024 00   A  0   0  4
  [ 2] .text             PROGBITS        00201000 002000 000bd6 00  AX  0   0  4
...
```

C'est le compilateur qui définit la section où doit être stocké chaque symbole, suivant ses caractéristiques (variable globale initialisée, non initialisée, fonction, ...), bien qu'il soit possible de forcer les symboles a être placés dans telle ou telle section (mot clef `.section` de l'assembleur, ou `__attribute__ ((section ("nom")))` de `gcc`). Dans `bootstrap/multiboot.S`, nous indiquons ainsi que la structure `multiboot_header` doit être placée dans la section `.multiboot`, le reste du code dans la section `.text`, et la pile de départ dans `.bss` (directive `.comm`).

En vérité, certaines de ces sections sont "virtuelles", dans le sens qu'elles ne correspondent à aucun morceau dans le fichier objet : elles ont une taille nulle sur disque, mais non nulle en mémoire. C'est le cas des sections du genre `.bss` : il s'agit de sections qui couvrent les variables globales non initialisées, dont la taille et les symboles lui appartenant sont calculés lors de la phase de compilation. Ce sera lors du chargement que de la place devra être allouée en mémoire pour cette section.

A.2 Scripts `ld`

Un script `ld` est un fichier au format texte qui n'a pas grand chose à voir avec un "script" au sens habituel. Il s'agit d'un fichier qui va *décrire* à `ld` où devront se situer les différentes sections des fichiers objet. Il permet également d'indiquer que tel symbole doit se situer à telle adresse, qu'à tel endroit en mémoire telle valeur doit être écrite au chargement du noyau, etc...

Ce sont des fichiers très répandus sous Unix. `/usr/lib/libc.so` en est souvent un par exemple, de même que `vmlinux.lds` dans les sources de Linux. Ils sont assez pratiques à manipuler, et assez méconnus. Pour SOS, le principal fichier de ce type est `support/sos.lds`. C'est ce fichier (voir la figure 13) qui contraint en particulier l'adresse à laquelle le premier octet du noyau devra être chargé : `0x200000`. C'est également ce fichier qui indique que la section `.multiboot` doit se situer au début du noyau, pour être sûr que la structure `multiboot_header` se situe dans les 8192 premiers octets imposés par le standard multiboot, puisque c'est en effet dans cette section que la structure a été définie (voir la section 4.2.2) :

```
/* our kernel is loaded at 0x200000 */
. = 0x200000;
__b_load   = .;

/* the multiboot header MUST come early enough in the output
   object file */
.multiboot :
{
    /* The multiboot section (containing the multiboot header)
       goes here */
    *(.multiboot);
 ...
```

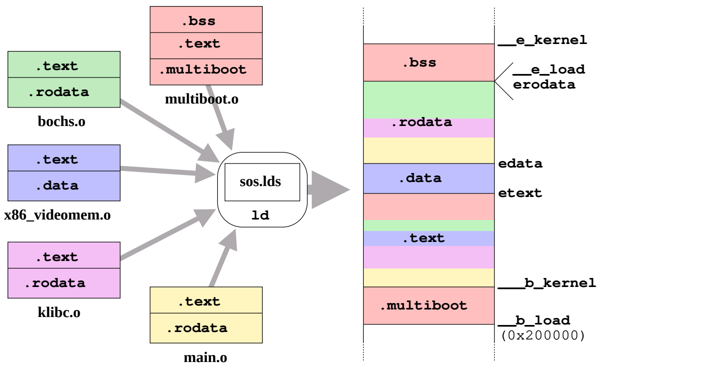


FIG. 13 – Effet du script `sos.lds`

Pour ce qui concerne le chargement de SOS par Grub, outre que ce fichier définit l'organisation en mémoire du noyau, il définit en plus les symboles `__b_kernel`, `__e_load` et `__e_kernel` qui sont référencés par le fichier objet `bootstrap/multiboot.o` :

```
...
/* We take note of the end of the data to load */
__e_load = .;

/* Beginning of the BSS section (global uninitialized data) */
.bss SIZEOF(.rodata) + ADDR(.rodata) :
{
    ...
    PROVIDE(_ebss = .);
}

/* We take note of the end of the kernel */
__e_kernel = .;
...
```

Pour le chargement par le secteur de boot fourni, le fichier `extra/sos_bsect.lds` possède le même rôle que `support/sos.lds`, et d'ailleurs il inclut directement son contenu. La différence est qu'il introduit la section `.bootsect` en préambule, qui contient le contenu de `extra/bootsect.o` (résultat de l'assemblage de `extra/bootsect.S`), et met en place la signature `0x55aa` en fin du secteur de boot, afin de pouvoir être chargé par le BIOS (voir la section 4.2.1). Il écrit également la taille du noyau à charger en nombre de secteurs (directive `LONG(...)`) en associant le symbole `load_size` à cette valeur ; il est intéressant de noter que cette valeur dépend des adresses que l'éditeur de liens générera pour le noyau.

Pour plus d'informations sur ces scripts, se reporter à la documentation de `ld` (`info ld`). Les scripts `ld` fournis devraient être suffisamment commentés pour être compris, une fois ces quelques explications données.

David Decotigny
`d2@enix.org` et `Thomas.Petazzoni@enix.org`
Site de SOS : `http://sos.enix.org`
Projet KOS : `http://kos.enix.org`

Á Johann Sebastian

Références

[1] Uresh Vahalia. *UNIX Internals : The New Frontiers*. Number ISSN 0131019082. Prentice Hall, 1995.

[2] Andrew Tanenbaum. *Systémes d'exploitation*. Number ISBN 2100045547. InterEditions / Prentice Hall, 1994.

[3] Patrick Cegielski. *Conception de systèmes d'exploitation – Le cas Linux*. Number ISBN 2212113609. Eyrolles, 2003.

[4] The Grub Team. *Multiboot Specification*. GNU, `http://www.gnu.org/software/grub/manual/multiboot/`, 2004.

[5] Intel Corp. Intel architecture developer's manual, vol 1, 1997.

[6] Jamie Lokier. Access to I/O-mapped / memory-mapped resources. `http://groups.google.com/groups?selm=fa.kdjtftv.182kca6%40ifi.uio.no`, 1998.

[7] PCI SIG. PCI local bus specification rev. 2.3, 2002.

[8] Alain Knaff. *Mtools*. GPL, `http://mtools.linux.lu/`, 2003.

[9] The Grub Team. *The GNU GRand Unified Bootloader*. GNU, `http://www.gnu.org/software/grub/`, 2004.

[10] The Bochs team. *The bochs IA-32/64 emulator*. LGPL, `http://bochs.sourceforge.net/`, 2004.

[11] Fabrice Bellard. *The qemu CPU emulator*. GPL, `http://fabrice.bellard.free.fr/qemu/`, 2004.

[12] Intel Compaq, Phoenix technologies. BIOS boot specification version 1.01, 1996.

[13] Intel Corp. Intel architecture developer's manual, vol 3, 1997.

[14] Ralf Brown. Ralf brown's interrupt (and port) list. `http://www.cs.cmu.edu/~ralf/files.html`, 2000.

[15] Xavier Leclercq. Reference of the BIOS interrupts. `http://www.xaff.org/GI/biosref.html`.

[16] Randall Hyde. Art of assembly language. `http://webster.cs.ucr.edu/AoA`.

[17] TIS Committee. TIS ELF. `www.x86.org/ftp/manuals/tools/elf.pdf`, 1995.