Cours Système n°5 : la mémoire

Éric Gaudefroy

École Hexagone
Cursus Cyberdéfense - M1

Les concepts exposés dans ce chapitre du cours système sont pour la plupart assez génériques, et applicables à la majorité des architectures de microprocesseur, mais les détails fournis correspondent aux architectures de type Intel/i386 (processeurs Intel 32 bits et compatibles) fonctionnant en mode protégé. Les principales différences avec d'autres architectures (ou d'autres modes de fonctionnement, par exemple le mode 64 bits des processeurs Intel et compatibles) sont exposées au fur et à mesure du document.

1. Aspects matériels

1.1. Généralités

1.1.1. Flots d'instructions CPU et interruptions

La succession des instructions machine exécutées par un CPU respecte une continuité logique. De manière schématique, le CPU exécute séquenciellement des instructions élémentaires stockées en mémoire, selon une progression linéaire aux sauts et appels de fonction près. Après l'exécution d'une instruction autre qu'un saut ou un appel de fonction (instructions qui spécifient explicitement l'adresse de l'instruction suivante, cf. cours sur l'assembleur), le CPU va chercher l'instruction suivante à l'adresse mémoire située immédiatement après l'instruction courante. Cette continuité n'est interrompue que :

- de façon asynchrone, par la réception d'une interruption matérielle (**interrupt**), qui est typiquement générée par un périphérique (IRQ, cf. */proc/interrupts* sous Linux) ;
- de façon synchrone :

  par la survenue d'une erreur lors de l'exécution d'une instruction machine (exception : division par zéro, point d'arrêt configuré par un debugger, etc.),

  par la réalisation volontaire d'une interruption logicielle par le code exécuté (trap).

Au cours de la séquence de démarrage de la machine, le BIOS, puis le noyau de l'OS, enregistrent dans une table (qui équivaut à un tableau de pointeurs de fonctions) leurs routines de traitement des interruptions. Lorsqu'une interruption (matérielle, logicielle, ou d'erreur) survient et qu'elle n'est pas masquée, le processeur interrompt le flot d'instructions en cours de traitement pour exécuter la fonction enregistrée, puis reprend le traitement principal au point où il avait été arrêté. En particulier, l'interruption « horloge » permet au noyau de faire fonctionner le système en mode multi-tâches (cf. le cours sur l'ordonnancement).

1.1.2. Accès mémoire et MMU

Le coeur du système est constitué par la mémoire (RAM [1]), qui stocke le code à exécuter et les données associées, et le CPU, qui réalise l'exécution du code. Il s'appuie sur le *chipset*, qui relie le CPU à la mémoire et aux périphériques. La mémoire est composée de « cellules » d'un octet accessibles individuellement ou par « mots » (blocs de 2, 4 ou éventuellement 8 octets) par le CPU. L'accès à une cellule ou un mot en mémoire se fait par son adresse, correspondant à un nombre identifiant de manière unique cette cellule (ou la première cellule du mot). La taille des adresses mémoire manipulables dépend du CPU : 16 bits, 32 bits, 64 bits, etc... Les architectures les plus répandues actuellement gèrent des adresses sur 32 bits (soit 4 Go de mémoire adressable) ou 64 bits [2].
L'unité de gestion de la mémoire (**MMU**, ou *Memory Management Unit*) est située sur la puce du CPU, en coupure de l'accès à la RAM. Elle réalise une traduction des adresses logiques utilisées par le code binaire de la tâche active en adresses physiques transmises sur le bus mémoire. Cette traduction permet, selon la configuration de la MMU, de rediriger les accès à une même adresse logique vers des adresses physiques différentes, et d'interdire tout accès à certaines plages de mémoire physique. Elle constitue ainsi le mécanisme matériel sur lequel est fondé le cloisonnement mémoire entre les différents processus du système d'exploitation, ainsi que la séparation entre l'espace mémoire noyau d'une part et la couche utilisateur d'autre part.
Cette traduction est transparente pour le code exécuté. Dans la mesure où toutes les tâches accèdent directement au CPU, indépendamment de leur niveau de privilèges (à l'exception éventuelle des « tâches invitées » lancées dans une machine virtuelle qui simule logiciellement un CPU), le rôle du noyau est de configurer convenablement la MMU avant de laisser la main au code en couche utilisateur, puis de rattraper et traiter les éventuelles exceptions. En particulier, le noyau n'a aucune action dynamique de filtrage ou de modification sur le code utilisateur dont il programme l'exécution.
Sur les architectures de type Intel 32 bits, la traduction effectuée par la MMU comprend deux étapes :

- la segmentation, qui transforme une adresse logique en adresse linéaire ;
- la pagination, qui traduit cette adresse linéaire en adresse physique.

Les mécanismes associés sont présentés dans la suite. Le mécanisme de pagination est, du point de vue conceptuel au moins, relativement générique, et se retrouve sur la plupart des architectures de CPU modernes.
La segmentation est en revanche très spécifique aux processeurs i386, et n'a en général pas d'équivalent sur les autres architectures. On notera que le terme d'adresse virtuelle est également souvent employé ; dans le cas des architecture Intel et compatibles, il recouvre les adresses logiques et linéaires, par opposition aux adresses physiques. Sur d'autres architectures, en l'absence de segmentation, la notion d'adresse linéaire n'existe pas et les adresses virtuelles sont synonymes des adresses logiques.

---

1. D'autres mémoires non volatiles s'y ajoutent, comme celle contenant le code du BIOS.
2. On notera cependant que la plupart des architectures 64 bits disponibles à ce jour ne prennent pas en compte l'ensemble des 64 bits d'une adresse, pour des raisons de complexité d'implémentation et de réalisme technique (il ne sert pas à grand chose à ce jour de pouvoir adresser 4 milliards de fois 4Go de mémoire physique, ou même virtuelle, sur une seule machine physique). Par exemple, l'architecture 64 bits AMD / Intel ne tient compte que de 48 bits dans les adresses virtuelles, et entre 36 et 48 bits, selon les modèles, dans les adresses physiques - soit entre 64 Go et 256 To physiquement adressables. Il n'est cependant pas exclu que ces plages soient étendues à l'avenir, dans de nouveaux modèles de CPU.

1.1.3. Mémoire et accès aux périphériques

L'accès à la mémoire et à certain périphériques à haut débit (carte graphique) se fait via un composant dit *Northbridge* qui peut être intégré directement au CPU ou appartenir au chipset. L'accès aux autres périphériques se fait via un composant plus lent du chipset : le *Southbridge*. Ces différents périphériques exposent des registres ou plages de mémoire pour leur communication avec le CPU. Cependant, tandis que l'accès à la mémoire principale se fait par adressage direct, la communication avec les autres périphériques nécessite la mise en oeuvre de mécanismes plus complexes, qui peuvent être de trois types :

- les ports d'entrées/sorties (PIO), qui appartiennent à un espace d'adressage distinct, sur 16 bits, et sont exploités par des instructions machine spécifiques (instructions assembleur **in** et **out**) ; voir */proc/ioports* sous Linux pour la liste des ports reconnus par le système ;
- les entrées/sorties projetées en mémoire (*MMIO*), pour lesquelles une mémoire de périphérique (par exemple, une RAM vidéo) est associée à une plage d'adresses mémoire physiques (il s'agit principalement des adresses hautes, situées au delà de la RAM principale) ; en fonction du résultat de la traduction de l'adresse par la *MMU*, un accès mémoire « standard » peut donc être redirigé vers un périphérique *MMIO* plutôt que vers la RAM ; voir */proc/iomem* sous Linux pour la liste des projections reconnues par le système ;
- les accès directs à la mémoire (**DMA**), qui offrent au périphérique la possibilité de lire ou d'écrire dans la RAM sans passer par le CPU [3] ; voir par exemple */proc/pci* et **lspci** pour voir quels sont les périphériques PCI qui sont capables de réaliser de tels transferts (« *Master Capable* ») [4] A la différence des accès PIO et MMIO, qui ne sont par construction réalisés qu'à l'initiative du CPU, un périphérique peut lui-même initier un transfert DMA, sans aucune interaction avec le CPU.

En plus de ces méthodes d'accès, les interactions entre le CPU et ses périphériques reposent largement sur les interruptions matérielles. Celles-ci fournissent aux périphériques un moyen asynchrone de signaler un événement au CPU, par exemple la réception d'un paquet par une interface réseau ou l'aboutissement d'un transfert DMA entre un contrôleur de disque et la mémoire principale.

1.2. Segmentation et privilèges

1.2.1. Privilèges CPU

A tout instant, le CPU associe au flot d'instructions en cours d'exécution un niveau de privilèges courant (*CPL*). Cette notion marque la distinction entre la couche utilisateur et le noyau du système d'exploitation. Il ne faut pas la confondre avec l'uid, utilisé par le noyau pour distinguer les privilèges des tâches en couche utilisateur vis-à-vis des appels système. Du point de vue du CPU, il n'y a donc pas de différence entre un processus *root* et un processus non privilégié. Inversement, c'est grâce aux privilèges CPU que le noyau peut s'imposer comme point de passage obligé pour l'établissement de communications entre processus ou avec les périphériques (et donc imposer l'utilisation d'appels système), et à ce titre réaliser efficacement les contrôles associés à l'uid [5]

3. Historiquement (bus ISA), les périphériques devaient passer par un contrôleur DMA qui réalisait pour eux l'accès RAM, et seuls les 16 premiers Mo de RAM étaient ainsi rendus accessibles ; voir */proc/dma* sous Linux pour la liste des « canaux » associés aux 2 contrôleurs DMA. Avec les bus récents (PCI), les périphériques peuvent utiliser le « bus mastering » pour prendre le contrôle du bus et effectuer par eux-mêmes des transferts arbitraires vers la RAM. Il est donc important de pouvoir faire confiance aux périphériques reliés à de tels bus.

4. Pour plus d'informations sur les périphériques, voir aussi **dmesg** et, sous Linux, /proc/ide/, /proc/bus/, /sys/ (système de fichiers virtuel sysfs du noyau 2.6), les utilitaires lsdev, lspci, lsusb et lshw, ainsi que hdparm et dmidecode.

5. Les privilèges de l'utilisateur "root" viennent donc :

Les architectures de type Intel fonctionnant dans le mode dit « protégé » définissent quatre niveaux de privilèges, ou *rings*, correspondant aux valeurs possibles du CPL :

- le ring 0, contenant le noyau du système d'exploitation (Linux, Windows ou autre), qui dispose de privilèges matériels maximaux ;
- les rings 1 et 2, en général non utilisés, qui peuvent servir à implémenter des surcouches noyau moins privilégiées ;
- le ring 3, contenant l'ensemble des tâches en couche utilisateur.

Le *ring* 0 est le seul à pouvoir utiliser les instructions de reconfiguration de la *MMU*, instructions qui ont un impact direct sur la gestion des privilèges CPU. Les rings 0 à 2 sont considérés comme faisant partie du noyau du système d'exploitation du point de vue des droits d'accès aux pages mémoire actives (cf. la section suivante). Les contrôles effectués par la *MMU* (essentiellement par la segmentation) permettent de limiter l'accès à la mémoire en fonction du ring. Un appel système consiste, pour un ring moins privilégié, à demander l'exécution d'une fonction dans un ring plus privilégié, implicitement supposé plus intègre et de confiance. Chaque *ring* peut accéder aux zones mémoire affectées aux rings moins privilégiés, ce qui permet notamment de lire ou d'écrire des données associées à un appel système.

Par ailleurs la régulation des entrées/sorties vers les périphériques s'effectue ainsi :

- pour les accès *PIO* :

  le CPU définit un niveau de privilèges d'entrées/sorties (*IOPL*), configurable en ring 0, qui correspond au dernier ring autorisé à masquer/autoriser les interruptions et à accéder aux ports d'entrée/sortie (si le *CPL* est inférieur ou égal à *l'IOPL*, l'accès est autorisé) ;

  il définit également un bitmap de permissions d'entrées/sorties permettant aux tâches de *CPL* plus élevé d'accéder à certains ports d'entrées/sorties.
- pour les accès *MMIO*, la régulation s'exerce par la gestion des droits mémoire associés aux adresses physiques concernées (segmentation et/ou pagination) ;
- le cloisonnement des transferts *DMA* repose essentiellement sur le fait que le noyau n'expose pas les interfaces (*PIO*, *MMIO*) permettant de configurer de tels transferts à la couche utilisateur [6].

Remarque : La très grande majorité des architectures CPU autres qu'Intel ne définissent que deux niveaux de privilèges CPU : un mode « superviseur » ou « superutilisateur » , correspondant au ring 0, et un mode « utilisateur » correspondant au ring 3. Les contraintes de portabilité sur de telles architectures expliquent largement le fait que, même sur une architecture Intel, les noyaux « classiques » (Linux, Windows, BSD) ne font aucun usage des rings 1 et 2. On notera également

- du "zèle" relatif du noyau lorsqu'il s'agit de répondre aux appels système issus d'un processus *root* (ouverture de *sockets* bas niveau, etc.) ;
- de la configuration des droits d'accès aux objets sensibles du système (*root* est propriétaire de nombreux fichiers de confifguration).

Le groupe *root* n'a en principe de privilèges qu'à travers ses permissions sur les fichiers. Il peut aussi y avoir des exécutables *root* avec bit « s » qui contrôlent explicitement le groupe avant de continuer : ainsi la commande **su** se restreint au groupe *wheel* sous BSD et certaines distributions Linux.

6. Ce cloisonnement vise à éviter qu'un utilisateur non privilégié ne se serve de tels transferts *DMA* pour contourner le cloisonnement mémoire, par exemple afin de modifier la mémoire noyau. Le mécanisme d'*IOMMU* présenté plus haut vise un objectif complémentaire de protection contre les périphériques malveillants

que la plupart des architectures CPU n'utilisent que très peu ou pas du tout les mécanismes de segmentation décrits plus bas, et associent directement la gestion des privilèges CPU au mécanisme de pagination (marquage « superutilisateur » ou « utilisateur » des pages contenant le code exécuté).

Les différents niveaux de privilèges décrits ci-dessus, ainsi que le mécanisme de pagination décrit plus bas, ne s'appliquent qu'au mode dit protégé des processeurs compatibles Intel 32 bits. Outre ce mode, qui est celui dans lequel fonctionnent la très grande majorité des systèmes d'exploitation, ces processeurs supportent trois autres modes de fonctionnement :

- le **mode réel**, mode de démarrage du processeur (typiquement utilisé par le BIOS et le bootloader), avec un adressage 16 bits et un seul niveau de privilèges équivalent au CPL 0 ;
- le **mode virtuel 8086**, mode utilisé pour la compatibilité avec des applications 16 bits (émulateur DOS par exemple), avec un adressage 16 bits (reposant également sur la pagination) et un seul niveau de privilèges correspondant au CPL 3 ;
- le **mode « System Management »** , mode dans lequel le processeur bascule sur réception de certaines interruptions, pour y exécuter du code spécifique typiquement intégré par le fabricant du processeur ; il s'agit d'un mode donnant accès à l'intégralité de l'espace mémoire avec un niveau de privilèges maximal.

Par ailleurs, les extensions 64 bits supportées par les processeurs récents ajoutent un mode supplémentaire, dit mode « long » , lui-même sous-divisé en un mode 64 bits et un mode 32 bits dit « de compatibilité » , utilisé pour l'exécution d'applications 32 bits par un système d'exploitation 64 bits, et largement similaire au mode protégé traditionnel. Ces différents modes de fonctionnement, et les transitions possibles entre eux, sont résumés dans la figure suivante.

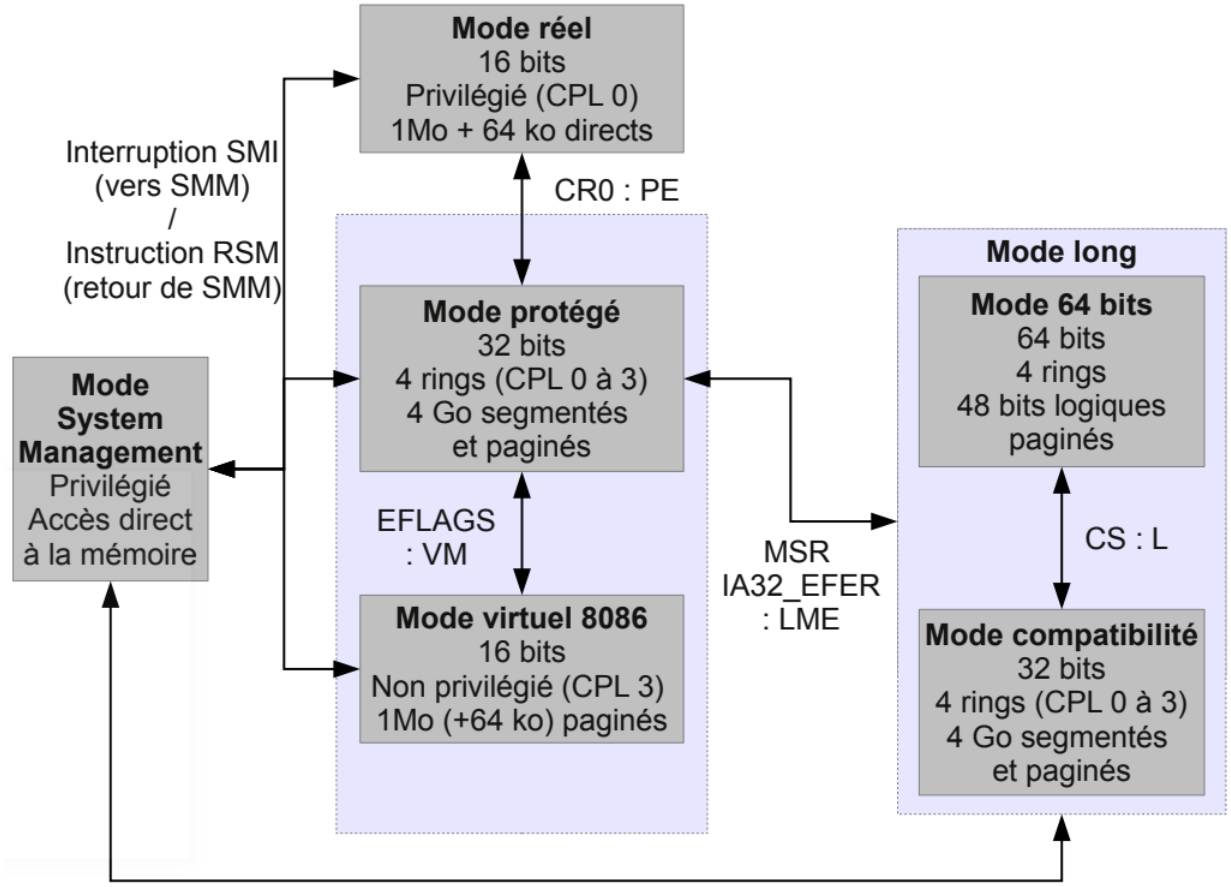

FIGURE 1 – Les modes de fonctionnement d'un processeur i386, avec extensions 64 bits (mode long).

1.2.2. Segmentation

Principe

La segmentation consiste à diviser l'espace des adresses linéaires (traductibles en adresses physiques par le mécanisme de pagination, voir la suite) en segments de taille ajustable, qui peuvent se recouvrir, et auxquels on affecte certaines propriétés, dont notamment :

- un type (code, données ou segment spécial) ;
- un domaine de visibilité (base, taille) ;
- un niveau de privilèges demandé (*DPL*) ;
- des restrictions spécifiques (code : lecture ; données : écriture).

Le *DPL* d'un segment de données correspond au *CPL* maximal requis pour pouvoir y accéder (en lecture et/ou écriture). Le *DPL* d'un segment de code correspond au *CPL* requis pour l'exécuter. Un appel système permet, moyennant un contrôle d'accès (voir plus bas), d'atteindre un segment de code de *DPL* inférieur au *CPL*. Ce *DPL* devient alors le nouveau *CPL* jusqu'au retour de l'appel système[7]. Une adresse logique complète est composée de l'adresse sur 32bits (ou offset), et d'une référence à un segment (qui peut prendre plusieurs formes). Sous réserve d'une autorisation de

7. Un segment de code peut aussi être déclaré conforme (hors programme), ce qui autorise un accès direct à son code depuis un ring moins privilégié mais sans changement de *CPL*.

l'accès (au regard du *DPL*, du type et des restrictions spécifiques), l'adresse linéaire est calculée comme suit :

- Adresse linéaire = Adresse logique + Base segment

si et seulement si :

- Type accès ≡ Type segment ;
- Adresse logique ≤ Taille segment ;
- $CPL \leq DPL$ segment ;
- Mode accès ⊆ Restrictions segment.

Lorsque la conversion n'est pas acceptable (CPL supérieur au DPL, type ou mode d'accès demandé incompatible du type ou des restrictions du segment, adresse supérieure à la taille du segment), une exception est remontée au CPU et peut être traitée par le système d'exploitation. Ce dernier traitera typiquement l'exception en déclenchant, par l'envoi d'un signal approprié (généralement SIGSEGV, cf. cours sur les signaux), la terminaison du processus ayant causé l'accès incorrect. Le cas particulier d'une exception remontée depuis le ring0, qui correspond à un accès mémoire incorrect par le noyau lui-même, entraîne en général un *kernel panic* nécessitant un redémarrage.

Mise en oeuvre

La liste des segments définis par le système est stockée dans deux tables, les tables globale (GDT) et locale (LDT) des descripteurs de segments. Chaque descripteur de segment contenu dans ces tables décrit, sur 64 bits, l'ensemble des propriétés d'un segment. Les deux tables ont sensiblement les mêmes caractéristiques, la GDT ayant plutôt vocation à être partagée par toutes les tâches, alors que la LDT peut être spécialisée en fonction de la tâche active. Dans la pratique, le *ring* 0 est libre de configurer et d'exploiter ces tables à sa façon. Les segments peuvent ainsi être référencés par des sélecteurs de segments, sur 16 bits, qui décrivent chacun un index dans l'une de ces deux tables [8].
Pour un accès mémoire donné, la référence au segment à utiliser pour la conversion en adresses linéaires peut être soit explicite (*far pointer*, composée d'un sélecteur de segment et d'une adresse dans le segment, cf. cours sur l'assembleur), soit implicite, par l'utilisation d'un sélecteur de segment par défaut. Les processeurs i386 permettent à cette fin de définir plusieurs sélecteurs de segments par défaut, inscrits dans des registres spécifiques du CPU, l'un ou l'autre de ces sélecteurs étant utilisé en l'absence de sélecteur explicite, en fonction du type d'accès mémoire :

- accès en exécution à du code (**fetch** de l'instruction suivante) via le segment CS, dont le sélecteur est stocké dans le registre %cs [9] ;
- accès en lecture ou écriture à des données via le segment DS (%ds) - en dehors des cas spécifiques décrits ci-dessous ;
- accès à des données de la pile via des instructions spécifiques (**push** et **pop**, cf. cours sur l'assembleur) via le segment SS (%ss) ;
- accès à des données dans le cadre d'une copie automatique (**rep movs** / **stos**) via le segment ES (%es).

8. Un sélecteur de segment permet également de définir un niveau de privilèges demandé, le *RPL* (hors programme), qui permet essentiellement à l'entité qui référence le segment de limiter ses prétentions en termes de niveau de privilèges pour l'accès.

9. Le RPL de ce sélecteur définit par ailleurs le CPL de la tâche en cours d'exécution.

Deux autres registres 16 bits, %fs et %gs, peuvent par ailleurs être utilisés pour stocker des sélecteurs de segments supplémentaires, à utiliser pour des références explicites aux segments qu'ils représentent.

Le mécanisme de segmentation ne peut être configuré que par le ring 0. Plus précisément, le contrôle d'accès repose sur les points suivants :

- seul le ring 0 est en mesure de définir la *GDT* et la *LDT* (instructions assembleur **lgdt** et **lldt**, réservées au ring 0), ce qui lui assure indirectement le contrôle des descripteurs de segment ;
- seul le ring 0 peut modifier le sélecteur CS contenu dans le registre %cs, et changer ainsi de *CPL* ; les transitions de *CPL* depuis les autres rings passent nécessairement par un mécanisme spécifique (interruption, appel système, cf. section suivante) sous le contrôle du ring 0 ;
- la modification des autres sélecteurs implicites (DS, SS, etc.) est autorisée depuis n'importe quel ring, mais ces sélecteurs n'ont aucune influence sur le *CPL* et ne permette que d'utiliser des segments prédéfinis par le ring 0 dans la *GDT* et la *LDT* et dont le *DPL* est supérieur ou égal au *CPL*.

La *MMU* se charge de traduire chaque accès à une adresse logique en une adresse linéaire, en effectuant les contrôles de privilèges et de mode comme décrit plus haut. L'identité entre adresse linéaire et adresse physique dépend de la présence ou non d'un mécanisme de pagination (qui n'est pas forcément activé dans le mode protégé du CPU, contrairement à la segmentation, obligatoire). Selon la façon dont le système choisit de tirer parti de ce mécanisme, la segmentation peut être réduite au seul cloisonnement entre couche noyau et couche utilisateur ou servir à séparer finement et de façon robuste les tâches et les zones mémoire qu'elles utilisent (code, données, librairies, pile, etc.). La plupart des systèmes d'exploitation se contentent du modèle de segmentation le plus basique (cf. le cas de Linux, plus bas) du fait de la plus grande flexibilité et de la simplicité de mise en œuvre offertes par le mécanisme de pagination, mais aussi pour des contraintes de facilité de portage sur des architectures ne disposant pas d'un mécanisme de segmentation aussi riche.

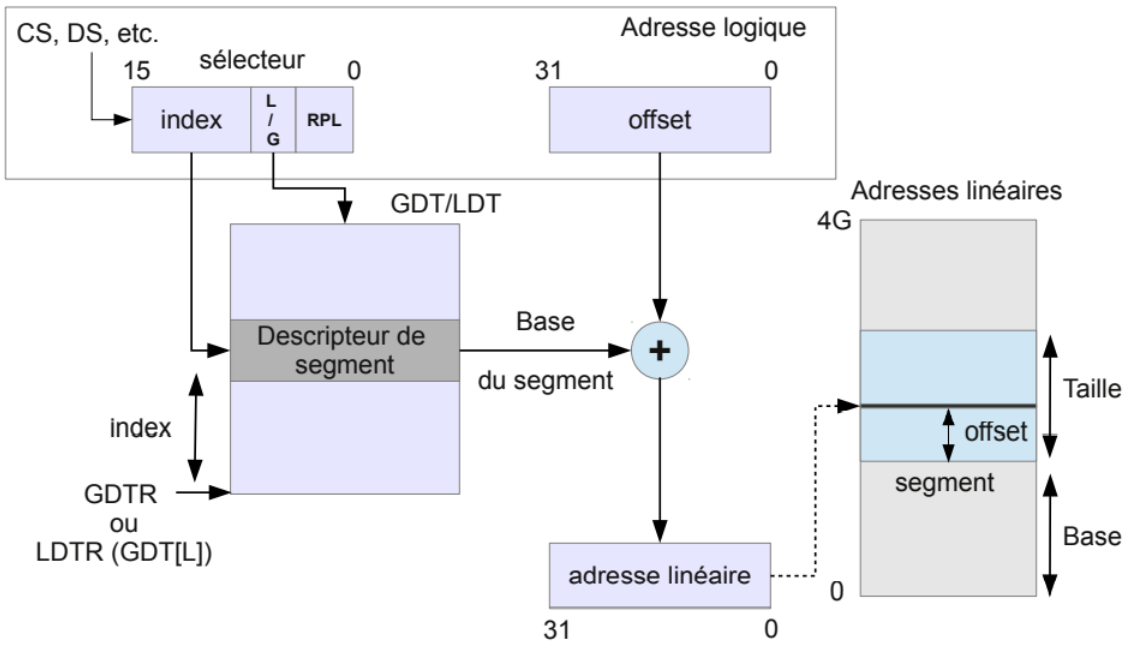

FIGURE 2 – Mise en oeuvre de la segmentation.

1.2.3. Changement de niveau de privilèges

Les changements de *CPL* se produisent :

- lors du traitement d'une interruption matérielle ou d'une exception ;
- lors d'un appel système ;
- au retour du traitement d'une interruption, d'une exception ou d'un appel système (restauration du *CPL* de la tâche interrompue).

L'appel système représente, au sein de la tâche courante, l'invocation d'une fonction privilégiée résidant dans l'espace noyau par une fonction de l'espace utilisateur. Une telle invocation ne peut se faire que sous certaines conditions (voir plus bas). Les architectures de type Intel/i386 prévoient trois méthodes pour réaliser un appel système :

- l'appel de fonction avec changement de segment, ou *long call* (mécanisme « historique ») ;
- l'interruption logicielle, ou *trap* (Linux/xBSD : interruption 0x80 ; Windows : interruption 0x2e) ;
- pour un processeur Intel, le couple d'instructions machine spécifiques SYSENTER et SYSEXIT (gérées par le noyau Linux 2.6 et Windows à partir de la version XP).

1.2.4. Segmentation sous Linux

Sous un Linux/i386 standard, la segmentation est essentiellement utilisée pour tracer la différence de *CPL* (*ring*) entre la couche noyau et la couche utilisateur. La *GDT* définit ainsi deux couples de segments :

- le premier couple, formé de KERNEL_CS (segment de code « rx ») et de KERNEL_DS (segment de données « rw »), représente l'espace noyau et est doté de privilèges maximaux (*DPL* 0) ; il couvre l'ensemble des adresses linéaires (0x00000000 - 0xffffffff) ;
- le second couple, formé de USER_CS (segment de code « rx ») et de USER_DS (segment de données « rw »), représente l'espace utilisateur et est doté de privilèges minimaux (*DPL* 3) ; il couvre également l'ensemble des adresses linéaires (0x00000000 - 0xffffffff).

Des segments supplémentaires sont utilisés par le noyau pour s'interfacer avec certaines fonctions du BIOS (*Advanced Power Management* et *Plug and Play BIOS*) ou mis à disposition de la couche utilisateur pour simplifier l'adressage des zones mémoire « privées » des threads (noyau Linux 2.6 et NPTL, cf. cours sur l'assembleur).

1.3. Pagination et swap

1.3.1. Limites fonctionnelles de la segmentation

Si on ne dispose que de la segmentation, l'allocation de la mémoire physique pour les tâches lancées se fait de façon contiguë par partitionnement de l'espace disponible. Se posent alors notamment les problèmes du dimensionnement des zones allouées à chaque tâche, de la réallocation de l'espace disponible et du recyclage des fragments libres isolés (défragmentation). Ce dernier point nécessite l'intervention d'un « ramasse-miettes » (*garbage collector*) capable de déplacer les entités présentes en mémoire physique au prix d'un ralentissement non déterministe des applications concernées. De façon générale, les stratégies d'allocation contiguë ne permettent jamais une utilisation optimale de la mémoire en termes de performances et de taux d'occupation.

La pagination apporte une réponse à ces problèmes. De plus, elle se prête naturellement à la gestion de zones mémoire partagées entre tâches et à l'extension virtuelle de la mémoire disponible (swap). Pour cette raison, la majorité des architectures CPU modernes ne proposent que des fonctions de pagination (en gérant toutefois deux « rings » pour distinguer entre couche noyau et couche utilisateur) et les systèmes d'exploitation pour CPU Intel s'appuyent davantage sur la pagination que sur la segmentation pour gérer les accès à la mémoire.

1.3.2. Principe de la pagination

La pagination consiste à découper l'espace des adresses linéaires (désegmentées) en pages (blocs contigüs de petite taille, typiquement 4ko) et la mémoire physique en blocs de même taille. La correspondance entre pages et blocs s'effectue par l'intermédiaire d'une table de pages. La table de pages est destinée à être modifiée à chaque changement de contexte (gestion d'une table par tâche), ce qui est intéressant si le système n'utilise pas ou peu la segmentation. Ce mécanisme permet, au cours de la vie des processus, d'allouer et de désallouer des pages sans contrainte de contiguïté en mémoire physique [10], d'où une utilisation optimale des ressources disponibles. Il permet également au noyau de gérer une mémoire virtuelle supérieure à la quantité de mémoire physique disponible (cf. swap, plus bas) et fournit un moyen supplémentaire (en plus de la segmentation) d'isoler les tâches. Pour chaque page de l'espace des adresses linéaires, la table de pages fournit typiquement les informations suivantes :

- la validité de la page (certaines adresses sont non valides - par exemple suite au « swap » du bloc de mémoire physique correspondant) ;
- le type de page (accès réservé au noyau ou autorisé en couche utilisateur) ;
- les permissions sur la page (une page valide peut, selon le type de CPU, être interdite en lecture, en écriture et/ou en exécution) [11] ;
- l'adresse du bloc physique correspondant à la page ;
- quelques directives spécifiques pour la gestion du cache.

Lorsqu'une tâche tente d'accéder à une page non valide ou selon un mode non autorisé, elle provoque une exception CPU (défaut de page). Cette exception est alors traitée par une routine du noyau, qui adapte sa réaction à la cause identifiée du problème :

- chargement de la page manquante (restauration d'une page swappée, chargement à la demande d'une portion de fichier mappé en mémoire ou d'une page vierge dans un mapping anonyme, cf. plus bas) ;
- duplication d'un bloc mémoire (pour la gestion du copy on write : séparation des données d'un processus père et de son fils lors d'une tentative d'accès en écriture, cf. plus bas) ;
- extension de la pile de la tâche appelante ;
- envoi d'un signal SIGSEGV (« Segmentation Fault ») [12] à la tâche dans les autres cas (adresse invalide ou accès selon un mode interdit).

10. On peut associer à deux pages contiguës dans l'espace des adresses linéaires deux blocs disjoints dans la mémoire physique.

11. Par défaut, sur une architecture i386 standard, les permissions possibles sur une page sont 0 (page invalide ou réservée au noyau), rx et rwx. Il faut utiliser la segmentation et/ou des extensions particulières (flag NX) pour obtenir des combinaisons différentes.

12. Selon les OS, le signal envoyé peut être SIGBUS (cas par exemple sous BSD) au lieu de SIGSEGV

Au retour du traitement de l'exception, la tâche reprend alors son exécution.

La traduction d'une adresse linéaire en adresse physique s'effectue de la façon suivante :

- les octets de poids fort de l'adresse linéaire représentent un numéro de page, et les octets de poids faible un offset dans la page ;
- la MMU utilise la table de pages associée à la tâche courante pour traduire le numéro de page en numéro de bloc ;
- l'adresse physique est alors obtenue en juxtaposant le numéro de bloc (poids fort) et l'offset (poids faible).

Dans la pratique, la table de pages est à deux ou trois niveaux. Les processeurs Intel/i386 (32 bits) ont par défaut deux niveaux de pagination.

La pagination permet d'isoler les processus si chacun dispose de ses propres tables de pages, stockées à des adresses différentes en mémoire. Le changement de contexte entre processus s'accompagne dans ce cas d'une modification du registre %cr3 pour le faire pointer sur le répertoire des tables de pages du nouveau processus. Une telle isolation est positive du point de vue de la sécurité, et limite les risques de propagation des incidents d'exploitation d'une tâche vers les autres tâches. Le noyau peut par ailleurs offrir des fonctionnalités de partage de zones mémoire en s'arrangeant pour référencer un même bloc dans les tables de pages de deux processus différents (à la même adresse linéaire on non). Les autres échanges doivent se faire de façon explicite via les appels système, ce qui revient à passer par une copie des données à échanger dans la mémoire noyau, qui est unique.

1.3.3. Pagination sous Linux

Sous Linux/i386, la mémoire est par défaut paginée à deux niveaux effectifs 16. Les tables de pages dépendent de la tâche active. La gestion des adresses linéaires s'effectue en général ainsi :

- les adresses 0x00000000-0xbfffffff (soit 3 Go d'espace adressable) sont utilisables par la couche utilisateur sous forme de pages standard de 4 Ko ; la présence et la nature des blocs mémoire en vis-à-vis de ces différentes adresses dépend de la tâche (processus) active (voir le chapitre suivant) ;
- les adresses 0xc0000000-0xffffffff (soit 1 Go d'espace adressable) sont sous le contrôle exclusif du noyau et ne dépendent pas de la tâche active.

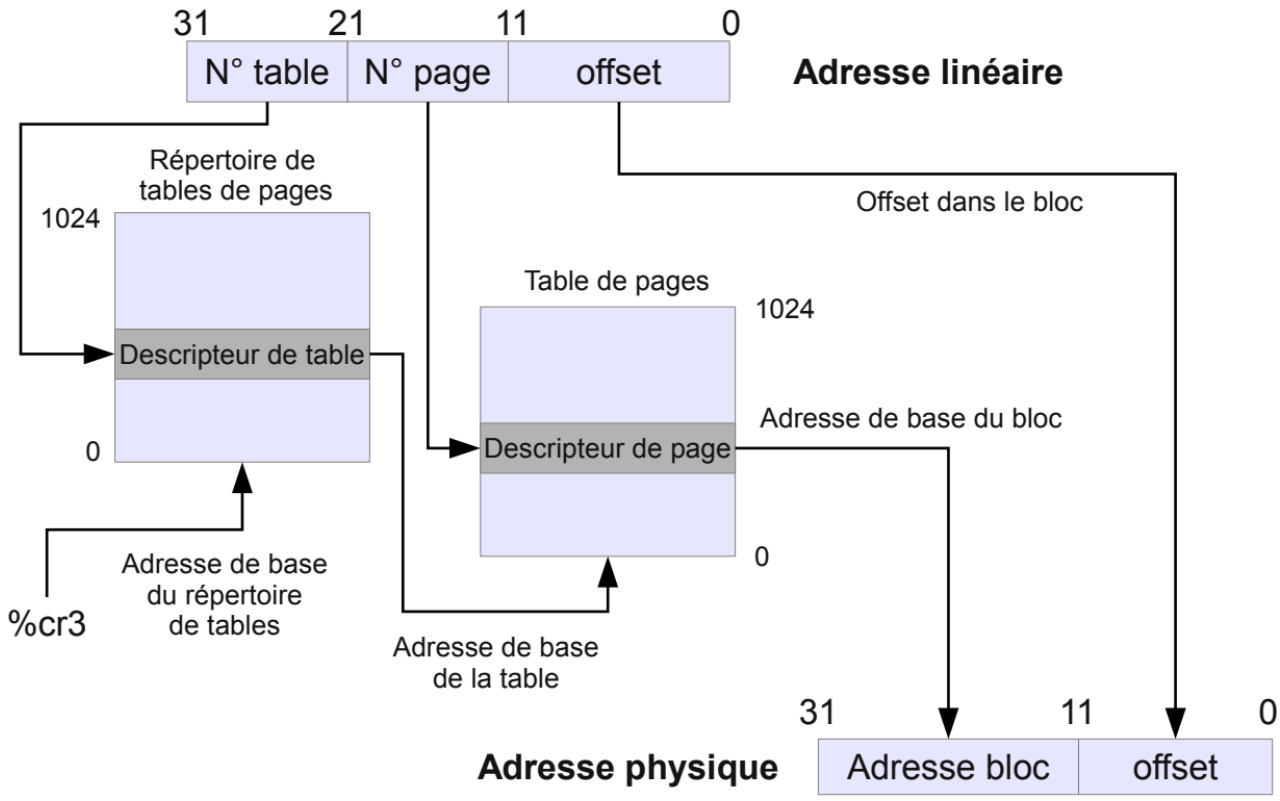

FIGURE 3 – Fonctionnement d'un mécanisme de pagination à deux étages.

1.3.4. Mémoire virtuelle (*swap*)

La gestion d'un mécanisme de swap permet d'augmenter virtuellement la quantité de RAM disponible, en sauvegardant le contenu d'un bloc de mémoire physique sur le disque dur, afin de permettre la réaffectation temporaire de ce bloc de mémoire à d'autres données, sans perdre le contenu d'origine. Ce mécanisme repose sur la pagination : les pages « swappées » sont marquées invalides dans la table de pages, de telle sorte que lorsque l'application tente d'accéder à une page non présente en RAM, le CPU génère une exception *page fault*, que le noyau traite en rechargeant la page depuis le disque, en se servant pour cela d'une table de correspondances entre numéros de pages et numéros de blocs disque dans le swap.
Selon le type de la page qui est swappée, son contenu sera sauvegardé sur le disque dur soit dans un fichier particulier (si la page appartient à une projection mémoire dite « partagée » , cf. section sur les projections mémoire), soit dans une partition de swap dédiée (dans tous les autres cas).

1.4. Caches

Les caches, ou antémémoires, représentent une stratégie duale et complémentaire de celle du swap :

- le mécanisme de swap consiste à utiliser un support plus large (mais plus lent) pour décharger les informations les moins utilisées et ainsi donner l'illusion d'une mémoire plus grande ;
- le mécanisme de cache consiste à utiliser un support plus rapide (mais plus étroit) pour stocker les informations les plus utilisées et ainsi donner l'illusion d'une mémoire plus rapide.

Les processeurs disposent ainsi de caches matériels pour accéder plus rapidement à la RAM : typiquement, un cache de niveau 1 (L1) sur la puce elle-même, un cache de niveau 2 (L2) également

sur la puce ou relié au processeur via un bus spécial rapide (pour des processeurs plus anciens) et dans certains cas un cache de niveau 3 (L3). Pour des raisons de performances (profils d'utilisation différents), les caches L1 séparent souvent un espace pour les instructions et un autre pour les données.

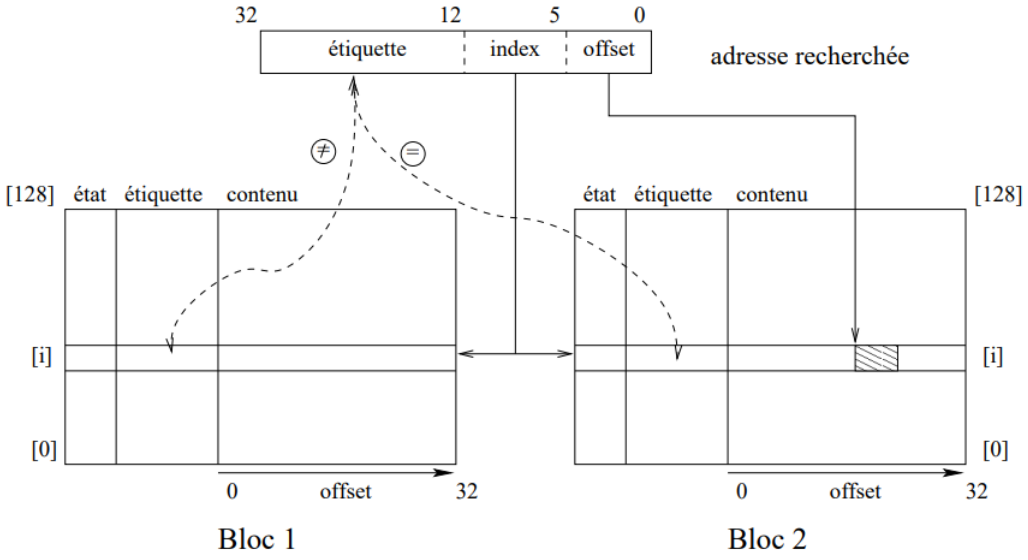

FIGURE 4 – Fonctionnement schématique du cache.

Le fonctionnement du cache est totalement transparent pour le code qui est en train de s'exécuter (mis à part les biais introduits dans les temps d'accès à la mémoire, cf. encadré ci-dessous). Ainsi, un accès en lecture à une adresse mémoire déclenche automatiquement la recherche de cette adresse dans le cache. Si l'adresse est présente dans une ligne de cache, la lecture est réalisée directement dans le cache, sans accès à la mémoire principale. Sinon, la lecture est faite en mémoire principale et son résultat est chargé dans le cache.

1.5. Hiérarchie des mémoires

Les différentes couches mémoire utilisées par une machine ont des caractéristiques différentes. Les couches les plus internes ont une capacité de stockage plus faible et des temps d'accès très courts, alors que les couches les plus externes permettent de stocker un très grand nombre d'informations mais avec des temps d'accès beaucoup plus longs. La hiérarchie correspondante est, de la mémoire la plus rapide à celle disposant des plus grandes capacités de stockage (et les moins chères au Ko) :

- les registres processeur, directement utilisables pour faire des calculs ou servant à stocker le contexte CPU d'une tâche (quelques dizaines d'octets) ;
- le cache matériel primaire (quelques dizaines de Ko) ;
- le cache matériel secondaire (quelques centaines de Ko) ;
- la mémoire principale RAM (quelques centaines de Mo à quelques Go) ;
- les disques durs locaux (quelques Go, voire To) ;
- les espaces de stockage distants (disques de serveurs de fichiers, etc., espace quasi illimité dans le cas d'Internet).

Du point de vue de l'utilisateur, l'objectif est de disposer de la capacité de stockage la plus importante avec les temps d'accès les plus courts.

Concrètement, les caches L1, L2 et L3 sont gérés de façon relativement transparente par le matériel (même s'il est possible de les désactiver et/ou de modifier leur comportement), le processeur n'accédant en principe pas directement à la RAM. Les registres sont chargés et déchargés depuis des adresses mémoire par le code exécuté (langage machine). Le disque local sert de mémoire virtuelle via les mécanismes de swap, gérés par le noyau du système d'exploitation. Il sert également à stocker les fichiers, auxquels accèdent les applications utilisatrices par des appels système – le noyau gérant en principe un cache en RAM pour ces fichiers (cache VFS). Enfin, les supports réseau sont importés par le noyau ou les applications réparties, qui peuvent gérer un cache sur le disque local (navigateur Web) et/ou en RAM.

2. Mémoire des processus

La mémoire associée à un processus en couche utilisateur se subdivise principalement en trois zones :

- le code (ou texte, car issu de la section .text de l'exécutable, cf. cours sur les exécutables), issu de l'exécutable chargé en mémoire (et des éventuelles bibliothèques dynamiques) ;
- les données, issues de l'exécutable et/ou gérées dynamiquement ;
- la pile, gérée dynamiquement.

Le chargement en RAM du contenu des fichiers (exécutable, librairies) s'effectue généralement à la demande, afin de diminuer le temps de lancement d'un programme. Le noyau ne charge pas immédiatement en mémoire physique le contenu d'un fichier projeté, mais conserve la liste des correspondances entre zones mémoire et fichiers.

L'espace mémoire d'un processus est composé d'un ensemble de zones appelées projections (*mappings*), elles-mêmes composées de pages mémoires associées à un fichier (code, librairies, etc.) ou non (pile, tas, etc.). La notion de projection se comprend bien dans le premier cas : le contenu d'une partie d'un fichier est projeté en mémoire, le contenu des pages correspondant à une copie de celui de la portion concernée du fichier. Ce même terme de projection est employé en l'absence d'association à un fichier, le mécanisme sous-jacent étant similaire (cf. section suivante sur la projection mémoire) - on parle dans ce cas de projection anonyme.
On trouve typiquement les zones suivantes :

- le code exécutable (et données en lecture seule), à l'adresse 0x08048000, r-xp, projeté depuis le programme exécutable ;
- les données initialisées : derrière le code, rw-p, projetées depuis le programme exécutable ;
- les librairies dynamiques et certains fichiers de paramétrage qu'elles projettent en mémoire (r–p), ainsi que les projections explicites par mmap (adresse par défaut) ;
- les données non initialisées et le tas : derrière les bibliothèques 22, rw-p (ou rwxp si un tas exécutable est demandé explicitement), anonymes ;
- la pile : située sous l'adresse 0xc0000000 (avec en général un offset négatif lié à la randomisation), rw-p ou rwxp, anonyme.

La construction de ces différentes zones mémoire au lancement d'un programme sera décrite plus en détail dans le cours sur les exécutables.

2.1. La zone de code

La zone dite de « code » contient au moins les éléments suivants (voir aussi le cours sur les exécutables) :

- le "texte" (.text) : code machine des différentes fonctions exécutables du programme ;
- les données en lecture seule (.rodata) : espace préinitialisé contenant les chaînes de caractères anonymes (chaînes de format des fprintf, etc.) et les variables globales ou statiques qualifiées par const.

La principale caractéristique de cette zone est d'être prévue pour fonctionner sans permission en écriture.

2.2. La zone de données

Les données d'un exécutable (ou d'une bibliothèque) se décomposent en deux catégories :

- les données initialisées (.data) : variables globales ou statiques, non constantes mais explicitement initialisées lors de leur déclaration, dont l'image mémoire est précalculée à la compilation et enregistrée dans le programme exécutable à la manière des .rodata ;
- les données non initialisées (.bss, pour Block Started by Symbol – l'acronyme n'est qu'historique) : variables globales ou statiques, non constantes et non initialisées au moment de leur déclaration, mises à zéro au chargement ;

A ces données définies dans l'exécutable lui-même, et allouées lors de son chargement, s'ajoutent celles qui sont allouées dynamiquement pendant la vie du programme. Le tas est une zone mémoire dédiée à de telles allocations dynamiques. Il est géré par le programme à travers les deux interfaces décrites dans les sections suivantes.

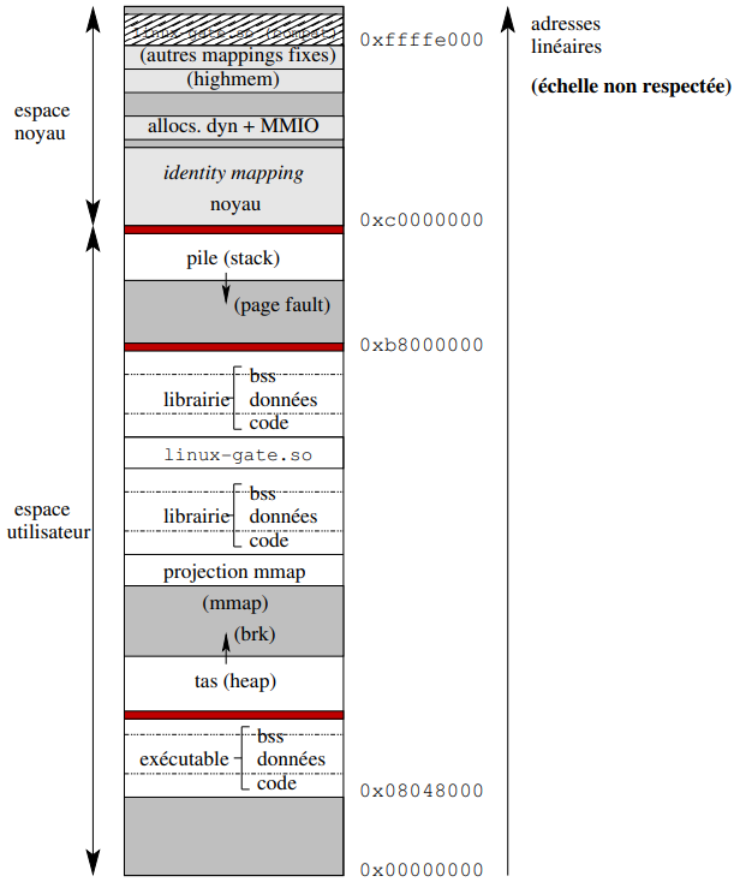

FIGURE 5 – Organisation en mémoire d'un processus (noyau Linux récent).

2.3. La pile

La pile (*stack*) est une zone de type « données » au sens de la séparation code/données. De taille variable, elle croît vers les adresses basses lorsqu'on y ajoute de nouvelles données. La pile est structurée dynamiquement par l'exécution des appels de fonction. A chaque appel de fonction correspond un étage de la pile (*stack frame*). Un étage de la pile contient :

- les éléments nécessaires à la suppression de l'étage courant et au retour à la fonction appelante ;
- les variables locales non statiques de la fonction ;
- les arguments d'appel à une sous-fonction, lorsque ceux-ci ne sont pas passés par des registres (cf. cours sur l'assembleur).

Lors du retour d'une fonction, l'étage correspondant est recyclé. C'est la raison pour laquelle il ne faut jamais renvoyer un pointeur sur une variable locale non statique. Le fonctionnement de la pile sera examiné plus en détail dans le cours sur l'assembleur. Contrairement au tas, qui est géré par la librairie standard, la gestion de la pile s'appuie pour beaucoup sur le travail du compilateur.
Le nombre de pages mémoire occupées par la pile (seule donnée tangible à l'échelle du système, qui travaille à une échelle moins fine que les déplacements du pointeur de sommet de la pile) est géré de façon transparente par le noyau, qui peut rallonger la pile lorsqu'il est prévenu d'un défaut de page au sommet.

2.4. Objets dynamiques

Les librairies dynamiques (et autres objets dynamiques) sont chargées à des adresses linéaires disponibles via des projections mémoire, avec une logique similaire aux projections associées à l'exécutable principal (sections de code, de données initialisées, et éventuellement de données non initialisées - le tas et la pile sont naturellement globaux et utilisés aussi bien par le code de l'exécutable que par celui des librairies). Ainsi, le chargement d'un exécutable lié dynamiquement (dépendant de librairies dynamiques) s'effectue selon le schéma suivant :

- le noyau charge (projette) l'exécutable ;
- le noyau voit qu'il est de type dynamique, il charge son interpréteur (éditeur dynamique de liens) ;
- l'interpréteur exploite les données relatives au chargement dynamique de l'exécutable, identifie et charge (projette) les dépendances ;
- l'interpréteur effectue l'édition dynamique de liens ;
- l'interpréteur repasse la main à l'exécutable.

Ce document est inspiré du « Cours Système n°5 : la mémoire » délivré par monsieur Vincent Strubel dans les cours dispensés à l'Agence nationale de la sécurité des systèmes d'information (Novembre 2012).