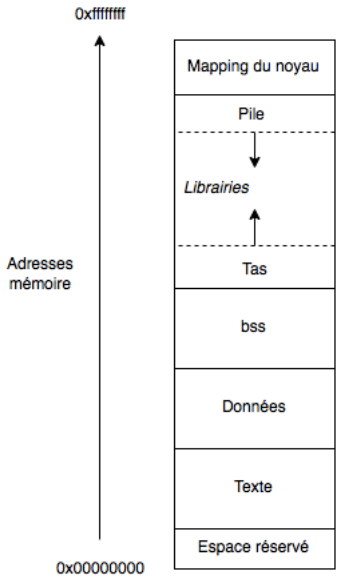
IT-201 TD3 - Elements de correction

baptiste.coye

February 2020

Organisation de l'espace virtuel des processus

1-a Organisation de la pile et du tas:



On organise de cette façon afin d'éviter les collisions entre tas et pile, ont met les librairie entre les deux parce qu'on a pas de place ailleurs.

1-b On remarque que le noyau est situé au dessus de l'espace user ce qui permet de garder le même espace virtuel pendant les appels systeme on a juste à changer la zone autorisée. Cf : Faille Meltdown

1-c On remarque que **les adressage user bougent de façon aléatoire** ce qui permet d'empêcher à un tier de comparer des executions différentes pour comprendre ce qu'il se passe dans le code. **Le noyau ne bouge pas** car dans tous les cas si quequ'un y a accès vous êtes foutus.

1-e Tant que les mallocs sont petits et qu'on a assez d'espace on utilise le tas alloué (on peut l'agrandir si nécessaire) après quand on a tout utilisé ou que le

malloc est trop gros pour passer le systeme alloue des nouveaux tas n'importe où dans l'espace virtuel et si on a plus de place on renvoie une erreur.

1-f L'implémentation est un peu complexe, il faut penser à créer une pile alternative pour pouvoir attraper le Signal envoyé lors de l'explosion de la pile. Contrairement au tas, la pile augmente linéairement avant d'exploser car on ne peut avoir de bouts disjoints. En pratique on est plutôt limité par le ulimit (8Mo) qu'on peut passer à unlimited mais ça va être long et on risque d'avoir remplit la ram et tué la machine avant de réussir à la faire exploser.

1-g Le but est de montrer de combien de bits utilisable dispose une architecture 64 bits. Pour cela on utilise le MAP_NORESERVE car d'habitude quand on veut allouer le noyau vérifie qu'on ait la place nécessaire dans ke swap. Quand on fait tourner le programme on remarque qu'on peut faire 131070 fois des allocations de 1Go ce qui représente 128To ce qui est exactement 2^{47}. On peut en conclure que nos machines 64 bits n'ont en fait que 47 bits utilisables. Dans /proc/cpuinfo vous pouvez voir desfois 48 bits affichés c'est que le noyau garde un peu d'espace au cas où pour lui.

1-h Si on mappe à l'adresse 0x1000 il nous place en 0x10000 donc un peu plus loin. Si on veut mapper sur la première page en 0x0, cette valeur est la même que NULL sauf que cette adressage est particulier car il veut dire "N'importe où". Si on veut mapper en 0x1 pour essayer de rester sur la première page, le noyau malheureusement aligne les mappings sur les débuts de pages pour éviter de perdre de la place. Donc 0x1 devient 0x0 ce qui veut dire n'importe où.

En pratique le noyau à une zone de 16 pages qui reste toujours invalide. Elle contient pleins d'adresses buggués qui nous permettent de segfaults au lieu d'avoir des corruptions memoire incompréhensibles.

Manipulation de la mémoire virtuelle

2-ab Pour faire rapide, lorsqu'on à un mapping privé et un mapping public qui sont utilisés en même temps, tant qu'on écrit sur le mapping public, les données sont partagés à l'ensemble des mappings sur une même page. Par contre au moment ou on écrit sur le mapping privé, ce dernier fait une copie de la page sur laquelle on a écrit. **Cette page contient les modifications qui avaient déjà étés faites par le mapping public** et modifie la page copiée. A partir de ce moment là **plus aucune modification du mapping public ne sera transmise au mapping privé**.

Du coup si on écrit 1111 en public puis 2222 en privé on obtiendra sur le mapping public 11110000 et 11112222 sur le mapping privé. Cependant si on écrit dans un premier temps 2222 sur le privé puis 1111 sur le public on aura obtiendra 00002222 sur le privé et 11110000 sur le public.

Dans les faits on utilise JAMAIS mapping privé et public en même temps sur une même page svp.

2-c Les défauts de pages arrivent juste lorsqu'une page n'est pas encore préchargée dans le cache. On remarque lorsqu'on fait tourner le programme qu'on a:

- Tout début = 145 défauts de page (Chargement du code)
- Open/Unlink = 147 défauts de page (Code de la lib C)
- Après le Mmap = 147
- Après une lecture = 148 (1 page faulté en plus)
- Après 4096 lectures = 148 (Toujours la même page)
- Après toute la lecture = 403 (255 autres pages)
- Après une écriture = 404 (Donc on fault en écrivant après lecture)
- Après toute l'écriture = 659 (255 autres pages)

Fun fact: Si on avait échangé l'ordre lecture et écriture on aurait eu 255 Défauts en moins car au moment de la lecture le système décide de passer la page en RO et ne la repasse qu'en RW qu'au moment de l'écriture. Si vous écrivez avant elle sera déjà préload en RW donc pourra être lisible sans le besoin de la recharger.

Fun fact 2: Si vous utilisez populate les pages sont passées en RO dès le Mmap donc On a 403 défauts au moment du Mmap pour les 147 de base + 256 de page de lecture.

Tables de pages

3-a On a un espace 32 bits, l'espace des adresses virtuelle est donc de 32 bits soit 2^{32} octets adressables. On a donc 4Go de mémoir virtuelle, sachant que les pages font 4096 octets on a 1 million de pages.

Les adresses étant codées sur 32 bits ils faut donc 4 octets pour stocker une adresse

3-b Chaque entrée de la table (PTE) contient $48 + 3 + 1 + 1 + 11 = 64bits$ Il faut donc 8 octets, sachant qu'il y a 1 million de page et une PTE par page il faut 8Mo pour stocker toutes les PTE.

3-c Le bit VALID permet de savoir si une page est valide... Pour savoir si une page va être utilisée dans le futur on utilise le principe de localité, si une page à été utilisée récemment elle a plus de chance d'être utilisée bientôt. Pour cela on utilise les **11 bits de AGE** dans lesquel le matériel met les heures de dernier access.

l'adresse physique est égale au nombre de bit du cadre physique ainsi que les bits permettants de trouver l'offset dans la page.

Ici on a 48 bits de cadre physique et des pages de 4ko soit 2^{12} on a donc un adressage physique de: $48 + 12 = 60$

En pratique nous n'avons pas de bits prédéfinis pour savoir si une page est présente en pratique on décide donc d'utiliser une valeur spéciale sur un Bit qui indiquera si la page est présente ou pas.

3-d La MMU n'ayant que très peu de mémoire, on ne peut clairement pas stocker un truc de 8Mo dedans. En mémoire physique c'est techniquement faisable mais 8Mo par process(Oui chaque process à besoin d'avoir la table des PTE) pour 200 processus on se retrouve avec 1,6Go de mémoire bouffée.

3-e On prend dans un premier temps les tableaux de niveau 1 et 2, chacuns ont 4096 octets libres, un pointeur faisant 4 octets (Cf 3-a) ont peut stocker 1024 adresses soit 2^{10} adresses. On utilise du coup 10 bits pour déterminer le pointeur à suivre dans ces niveaux.

Au niveau 3, on peut mettre 512 PTE (8 octets cf 3-b) par tableau donc on utilise 9 bits pour déterminer le PTE à utiliser.

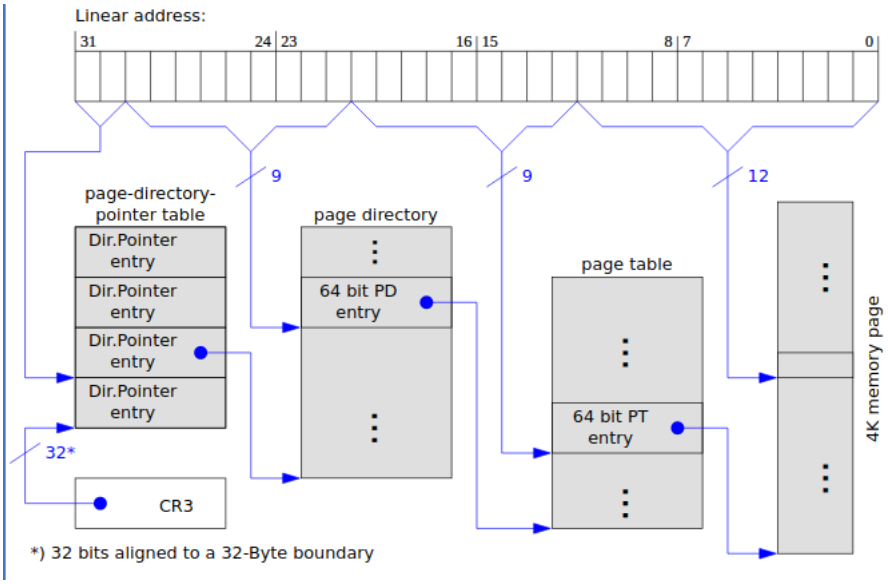
à la fin de l'adresse on veut savoir l'offset dans la page soit 12 bits (cf 3c)

On a donc un adressage constitué de:

- 10 bits pour le niveau 1
- 10 bits pour le niveau 2
- 9 bits pour le niveau 3
- 12 bits pour l'offset dans la page.

Soit un adressage sur 41 bits or on en a que 32 du coup il faut réduire le nombre d'addressages possible, le plus simple étant de réduire le niveau 1 c'est ce qu'on fait, on le limite à 1 bit au lieu de 10.

ça nous donne un schéma de ce style là (Ici on a le niveau 1 qui fait 2 bits au lieu de 1)



3-f L'avantage de fonctionner de cette façon est qu'**on à pas à allouer systématiquement les 8Mo de l'adressage linéaire**. On peut faire en fonction de nos besoins.

Exemple 1ko sur une seule page:

- 1 PTE (8 octets)
- 1 tableau de niveau 3 (4ko)
- 1 pointeur de niveau 2 (4 octets)
- 1 tableau de niveau 2 (4ko)
- 1 pointeur de niveau 1 (4 octets)
- 1 tableau de niveau 1 (4 ko)

on se retrouve donc avec 12ko + miettes
Si on le refait pour 1Go (soit 250 000 pages de 4ko)

- 250 000 PTE
- 500 tableaux de nivaeu 3
- 500 pointeurs de niveau 2
- 1 tableau de niveau 2
- 1 pointeur de niveau 1
- 1 tableau de niveau 1

En négligeant les pointeurs et le PTE on obtient $(500 + 1 + 1) * 4ko = 2Mo$