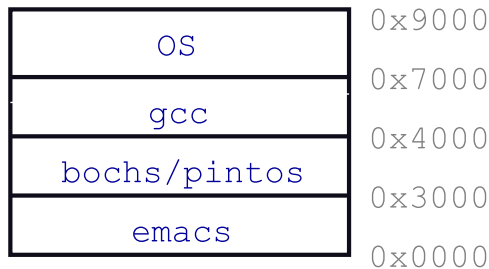


# SEA: Mémoire Virtuelle

Instructor: Pablo Oliveira

ISTY

## Problèmes de l'adressage direct



- Que faire si gcc souhaite plus de mémoire ?
- Si emacs souhaite 5 Go de mémoire sur une machine qui possède 4Go ?
- Si gcc écrit par erreur sur l'adresse 0x7100 ?
- Est-ce que le compilateur/linqueur doit savoir que gcc est à l'adresse 0x4000 ?
- Que faire si un processus veut libérer sa zone mémoire ?

# Problèmes liés au partage de la mémoire physique

- Protection

- Un bug dans un processus peut corrompre un autre
- Protéger les écritures de  $A$  dans la mémoire de  $B$
- Protéger la lecture de la mémoire de  $B$  (espionner mots de passe) (ssh-agent)

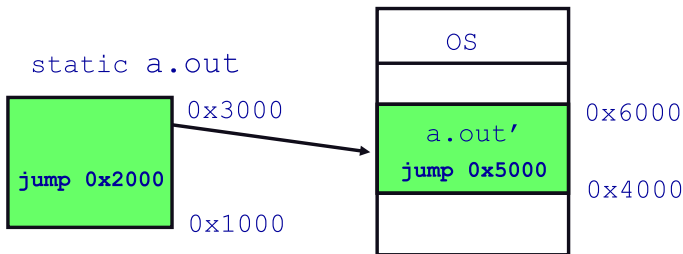
- Transparence

- Un processus ne doit pas exiger des positions fixes en mémoire

- Utilisation efficace de l'espace mémoire

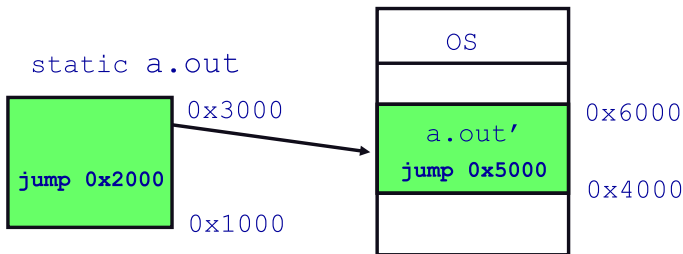
- Mémoire totale des processus souvent dépasse la mémoire physique de la machine.

## Idée : linkeur à la volée ?



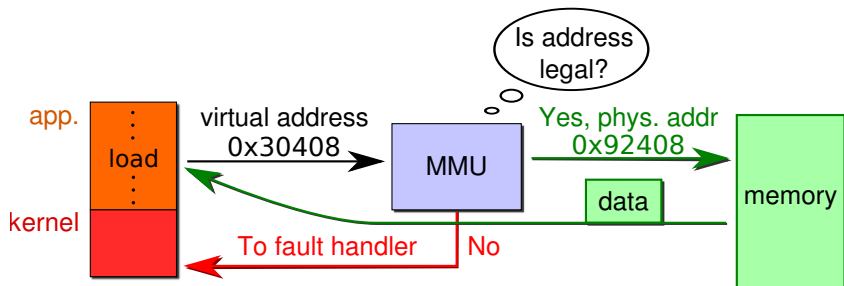
- *Linkeur* patche les adresses des symboles
- Idée : fait le lien juste avant l'exécution (pas à la compilation)
  - Détermine où les processus seront chargés (base)
  - Ajuste toutes les adresses (par addition de la base)
- Problèmes ?

## Idée : linkeur à la volée ?



- *Linkeur* patche les adresses des symboles
- Idée : fait le lien juste avant l'exécution (pas à la compilation)
  - Détermine où les processus seront chargés (base)
  - Ajuste toutes les adresses (par addition de la base)
- Problèmes ?
  - Comment mettre en place la protection ?
  - Comment faire la migration (pointeurs) ?
  - Nécessite un espace contigu suffisamment grand.

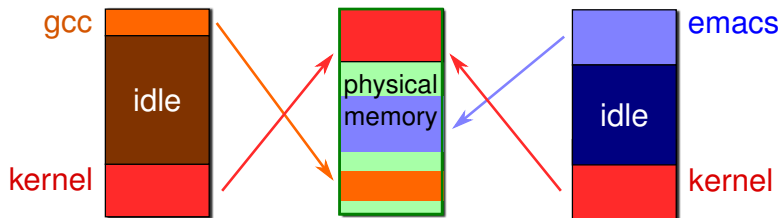
# Mémoire Virtuelle



- Chaque processus à son propre espace de mémoire “virtuelle”
  - La MMU (Memory-Management Unit) traduit les adresses virtuelles en adresses physiques lors de chaque lecture ou écriture.
  - L’application n’a jamais accès à la mémoire physique.
- Protège l’accès à la mémoire
  - Un processus ne peut pas accéder à la mémoire d’un autre processus
- La mémoire virtuelle peut dépasser la mémoire physique disponible

# Avantages de la mémoire virtuelle

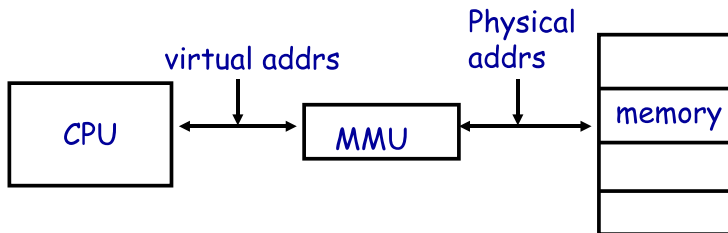
- Supporte la migration dans l'espace mémoire
  - Une partie des pages est dans la RAM, une autre partie sur disque.
- La majorité de la mémoire d'un processus est inactive (règle du 80/20).



- Pages inactives sont sauvegardées sur disque
- D'autres processus peuvent récupérer la mémoire libérée
- Semblable à la virtualization CPU : processus n'utilise pas le CPU → préemption.
- Inconvénient : MV = indirection → ralentissement

# Définitions

- Les programmes écrivent sur des **adresses virtuelles** (or **logiques**)
- La mémoire réelle utilise des **adresses physiques** (ou **réelles**)
- Le matériel qui fait la traduction est la Memory Management Unit (**MMU**)

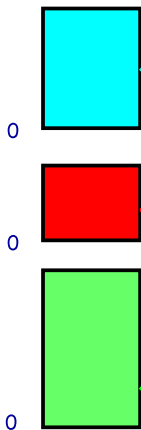


- Inclue dans le CPU
- Configurée en ring 0 (e.g., registres base et borne)
- Donne à chaque processus un **espace d'adressage** virtuel.

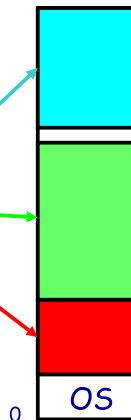


# Espace d'adressage

Virtual Address View



Physical Address View



## Idée : registres base + borne

- Deux registres spéciaux utilisés par la MMU : **base** et **borne**
- Pour chaque écriture/lecture :
  - Adresse Physique = Adresse Virtuelle + **base**
  - On vérifie  $0 \leq \text{adr.virtuelle} < \text{borne}$ , sinon interruption.
- Comment déplacer un processus en mémoire ?
- Que faire lors d'un changement de contexte ?

## Idée : registres base + borne

- Deux registres spéciaux utilisés par la MMU : **base** et **borne**
- Pour chaque écriture/lecture :
  - Adresse Physique = Adresse Virtuelle + **base**
  - On vérifie  $0 \leq \text{adr.virtuelle} < \text{borne}$ , sinon interruption.
- Comment déplacer un processus en mémoire ?
  - Change le registre **base**
- Que faire lors d'un changement de contexte ?

## Idée : registres base + borne

- Deux registres spéciaux utilisés par la MMU : **base** et **borne**
- Pour chaque écriture/lecture :
  - Adresse Physique = Adresse Virtuelle + **base**
  - On vérifie  $0 \leq \text{adr.virtuelle} < \text{borne}$ , sinon interruption.
- Comment déplacer un processus en mémoire ?
  - Change le registre **base**
- Que faire lors d'un changement de contexte ?
  - SE doit recharger les registres **base** et **borne**

# Avantages et Inconvénients du système Base+Borne

- Avantages
  - Matériel simple : 2 registres, un additionneur et un comparateur
  - Rapide : quelques cycles seulement pour faire la traduction
  - Exemple : Cray-1 utilisait un système Base + Borne
- Inconvénients

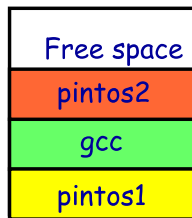
# Avantages et Inconvénients du système Base+Borne

- Avantages

- Matériel simple : 2 registres, un additionneur et un comparateur
- Rapide : quelques cycles seulement pour faire la traduction
- Exemple : Cray-1 utilisait un système Base + Borne

- Inconvénients

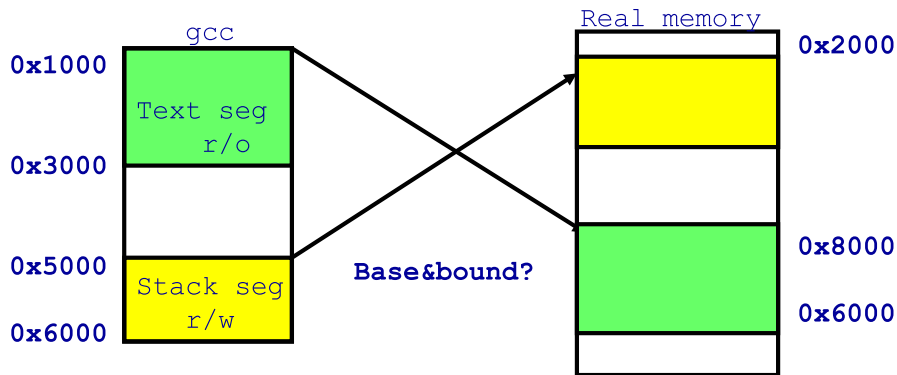
- La mémoire d'un processus doit être contigue
- Pas de mémoire partagée entre processus



- Une solution : segments multiples

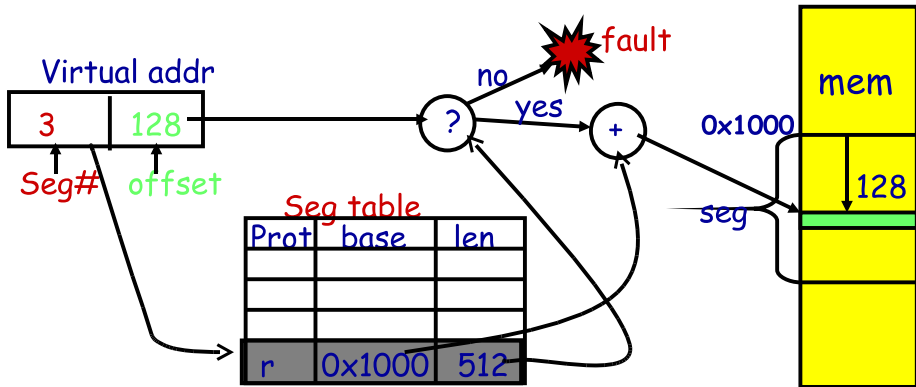
- E.g., on sépare le code, la pile et les données
- Eventuellement plusieurs segments de données

# Segmentation



- Chaque processus dispose de plusieurs registres base/borne
  - Espace d'adressage dispose de plusieurs segments
  - Protection mémoire par segment
- Chaque accès mémoire doit spécifier le segment accédé

# Implémentation de la segmentation

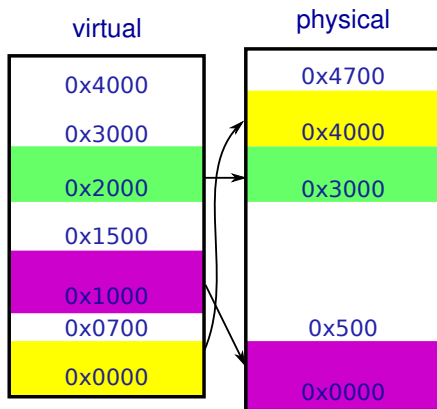


- Chaque processus dispose d'une table de segments
- Chaque AV est composée d'un segment et d'un offset
  - Bits de poids fort donnent le segment, Bits de poids faible donnent l'offset (PDP-10)
  - Où alors segment choisi par l'instruction utilisée (x86)



## Exemple de Segmentation

Seg	base	bounds	rw
0	0x4000	0x6fff	10
1	0x0000	0x4fff	11
2	0x3000	0xffff	11
3			00



- Numéro de segment sur 4-bits (premier chiffre), offset sur 12 bits (3 derniers chiffres)
  - Où est 0x0240 ? 0x1108 ? 0x265c ? 0x3002 ? 0x1600 ?

# Avantages et Inconvénients de la segmentation

- Avantages

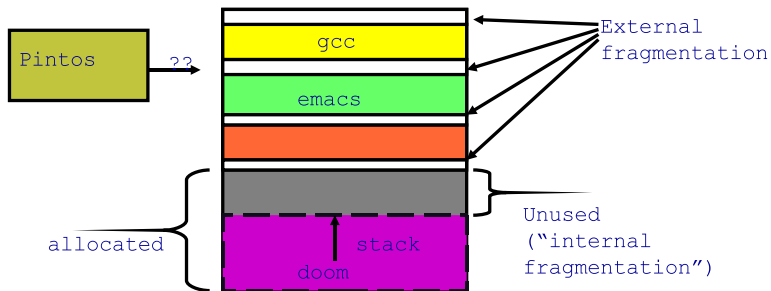
- Plusieurs segments par processus
- Permet le partage (Comment ?)
- La mémoire du processus peut-être partiellement sur disque.

- Inconvénients

- Surcoût d'accès à la table des segments
- Segments ne sont pas transparents pour le programme (instructions nécessaires pour choisir le segment)
- Segment de taille  $n$  nécessite  $n$  octets de mémoire *contigue*
- Problème de *Fragmentation*

# Fragmentation

- **Fragmentation**  $\implies$  Mémoire libre mais inutilisable
- Après un certain temps :
  - Segments de taille variable = plein de petits trous (fragmentation externe)
  - Segments de taille fixe = pas de trous externes, mais segments sous-utilisés (fragmentation interne)



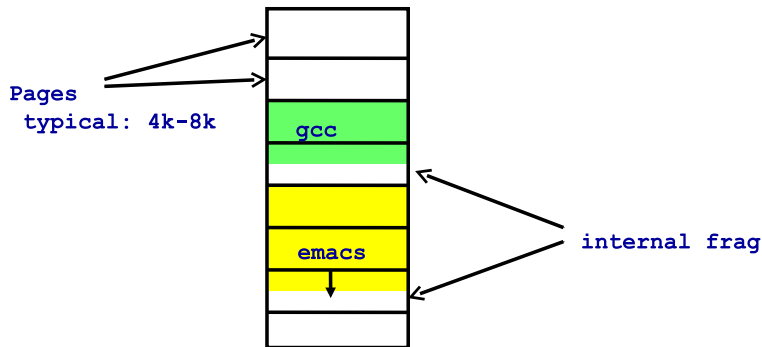
# Alternatives à la MMU

- Protection au niveau du langage (Java)
  - Plusieurs modules se partagent le même espace d'adressage
  - Le langage garantit l'isolation
- Gardes générées au niveau du compilateur
  - Le compilateur émet des vérifications avant chaque écriture/lecture
  - Google **Native Client** utilise cette méthode.

# Pagination

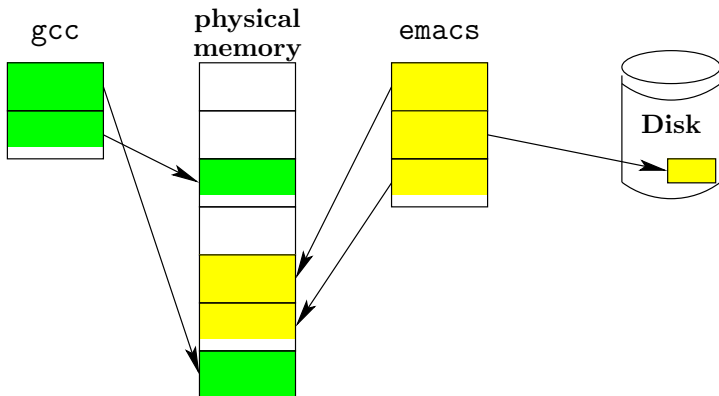
- On divise la mémoire en petites pages (4K)
- Chaque page physique est associée à une page virtuelle
  - La table d'association est propre à chaque processus
- Protection à la granularité d'une page
  - Page lecture seule (interruption)
  - Page invalide (interruption)
  - Le SE peut changer le mapping et revenir à l'application (chargement à la demande)

# Avantages et Inconvénients de la pagination



- Pas de fragmentation externe
- Implémentation simple (allocation, free et swap). Les pages d'un même segment ne sont pas forcément contigues.
- En moyenne chaque segment mémoire gâche une demi page.

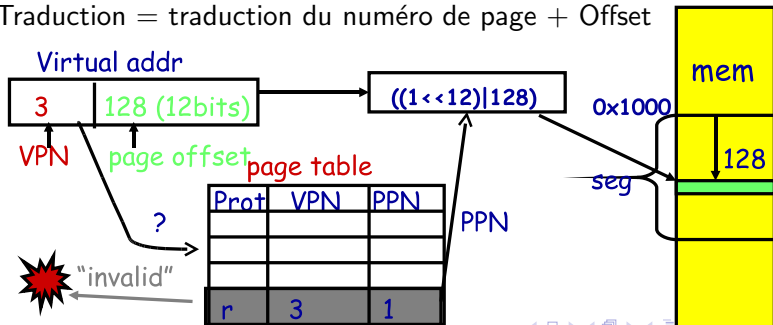
# Allocation simple



- Alloue n'importe quelle page physique libre (pas forcément contigues)
- Les pages inactives peuvent être stockées sur disque

# Implémentation de la pagination

- Pages de taille fixe (souvent 4K)
  - 12 bits de poids faible ( $\log_2 4K$ ) pour l'offset
  - bits de poids fort sont le numéro de page
- Chaque processus possède une table des pages
  - Traduit les numéros de page virtuels en numéros de page physiques
  - Des informations supplémentaires sur les protections, droits, etc.
- Traduction = traduction du numéro de page + Offset





# Quelle est la taille de la table des pages ?

- Page de 4K
- Adresse sur 32 bits (4Go)
- Nombre de pages =  $2^{32}/4096 = 1.048.576$
- Problème ?

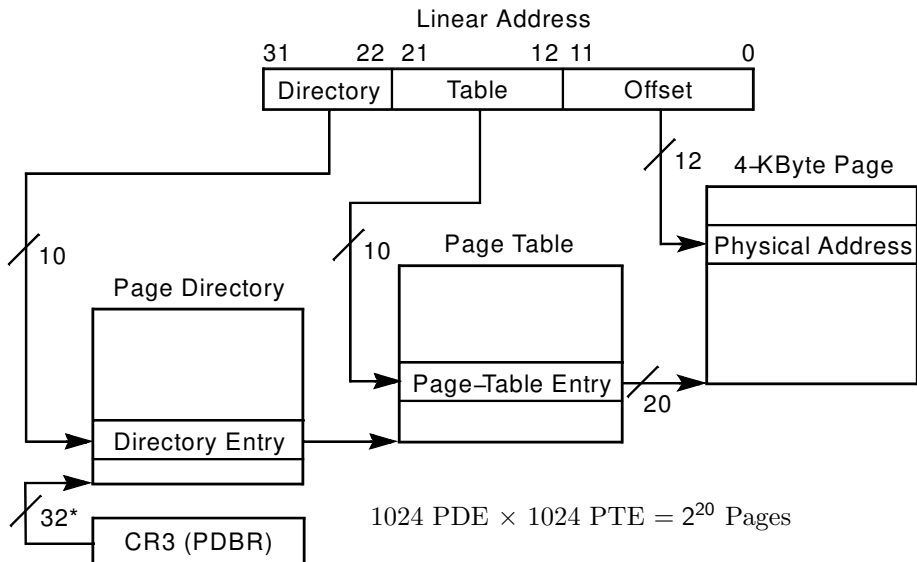
# Quelle est la taille de la table des pages ?

- Page de 4K
- Adresse sur 32 bits (4Go)
- Nombre de pages =  $2^{32}/4096 = 1.048.576$
- Problème ?
  - Il faut plusieurs Mo pour stocker la table des pages de *chaque processus*!
  - Table des pages hiérarchique

# Pagination sur x86 : table de pages hiérarchique

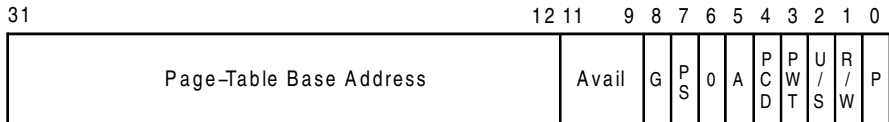
- Pagination activée grâce à un registre de contrôle (%cr0)
  - L'écriture de ce registre nécessite le mode privilégié
- Souvent page 4K
- %cr3 : pointe vers le répertoire des tables
- Répertoire des tables : 1024 entrées
  - Chaque entrée pointe vers une table de pages
- Table des pages : 1024 entrées
  - Chaque entrée donne la traduction d'une page de 4K
  - Chaque table est donc en charge de 4Mo de mémoire virtuelle

# Traduction sur x86



# Répertoire sur x86

Page-Directory Entry (4-KByte Page Table)



Available for system programmer's use

Global page (Ignored)

Page size (0 indicates 4 KBytes)

Reserved (set to 0)

Accessed

Cache disabled

Write-through

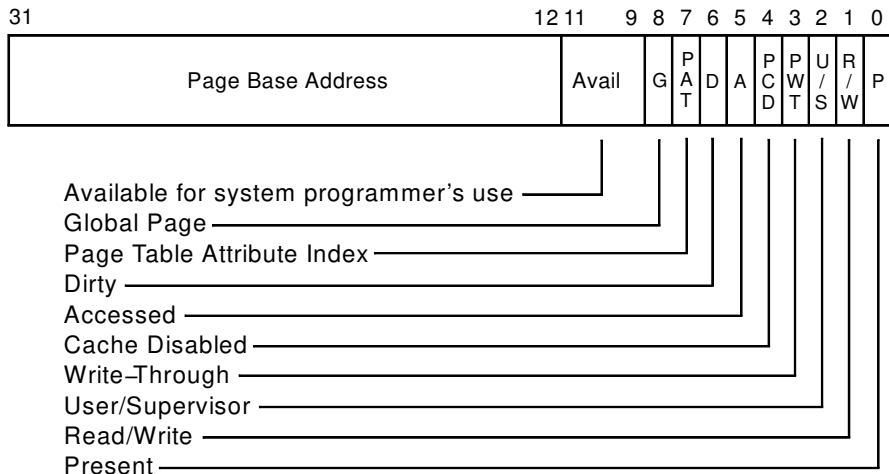
User/Supervisor

Read/Write

Present

# Table des pages sur x86

Page-Table Entry (4-KByte Page)



# Coût de la Pagination : comment la rendre efficace ?

- Traduction sur x86 nécessite trois accès par lecture/écriture :
  - Lecture de l'entrée dans le répertoire
  - Lecture de l'entrée dans la table des pages
  - Lecture de l'adresse initiale après traduction

# Coût de la Pagination : comment la rendre efficace ?

- Traduction sur x86 nécessite trois accès par lecture/écriture :
  - Lecture de l'entrée dans le répertoire
  - Lecture de l'entrée dans la table des pages
  - Lecture de l'adresse initiale après traduction
- Pour être efficace le CPU cache les traductions récentes
  - *Translation Lookaside Buffer* or **TLB**
  - Chaque TLB contient les dernières entrées de page accédées
  - Configurations typiques : 64-2K entrées, 4-way to fully associative, 95% hit rate
- Pour chaque accès
  - Si l'adresse est dans le TLB, traduction directe
  - Sinon parcours du répertoire de pages et stockage dans le TLB pour les prochains accès

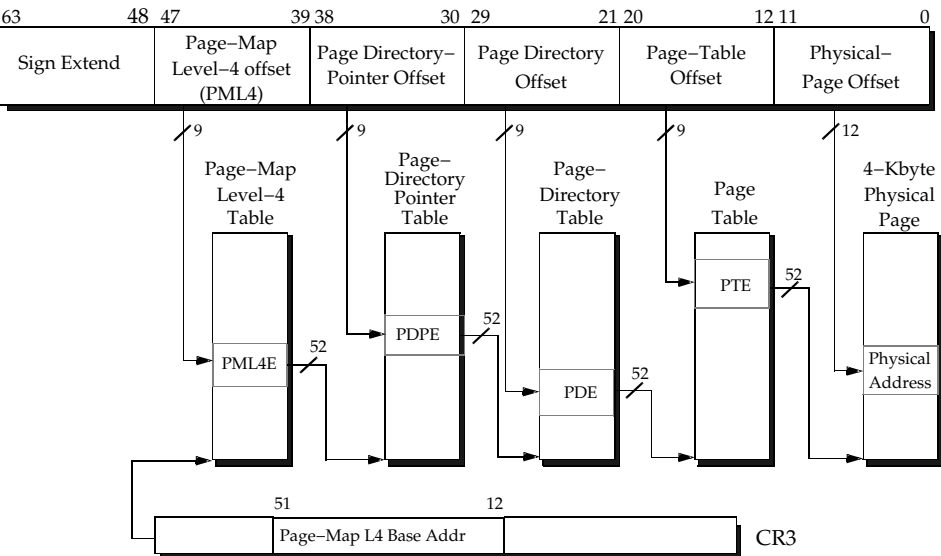


# TLB details

- TLB opère directement sur le pipeline CPU  $\implies$  rapide
- Que se passe t'il lors d'un changement de contexte ?
  - Flush TLB
  - Chaque entrée est taggée avec un PID
- C'est le rôle du SE de maintenir le TLB valide
- E.g., x86 instruction *invlpg*
  - Invalide une entrée TLB

# x86 long mode paging

Virtual Address



# Espace d'adressage du SE

- Son propre espace ?
  - Impossible : sur de nombreux machines un appel système ne change pas les tables de pages
  - Rendrai plus difficile le passage de pointeurs à un appel système
- Donc OS dans le même espace d'adresse que le processus
  - Utilise la protection des pages pour protéger la zone mémoire du SE

# Avantages de la pagination

- Chargement à la demande
- Augmenter la taille de la pile
- Allocation des pages BSS
- Données et bibliothèques partagées
- Pages partagées
- Copy-on-write (`fork`, `mmap`, etc.)