Gestion de la mémoire

Adresses

Allocation mémoire contiguë

Segmentation

Pagination

Structure des tables de pages

Mémoire centrale

Seule mémoire accessible directement par le CPU

Les caches du processeur sont des “détails d’implantation”

Partagée entre plusieurs processus

- Partage efficace et pratique: pas besoin de coopération explicite
- Protection mutuelle: pas d’interférence inattendue

Adresses *physiques* vs adresses *logiques*

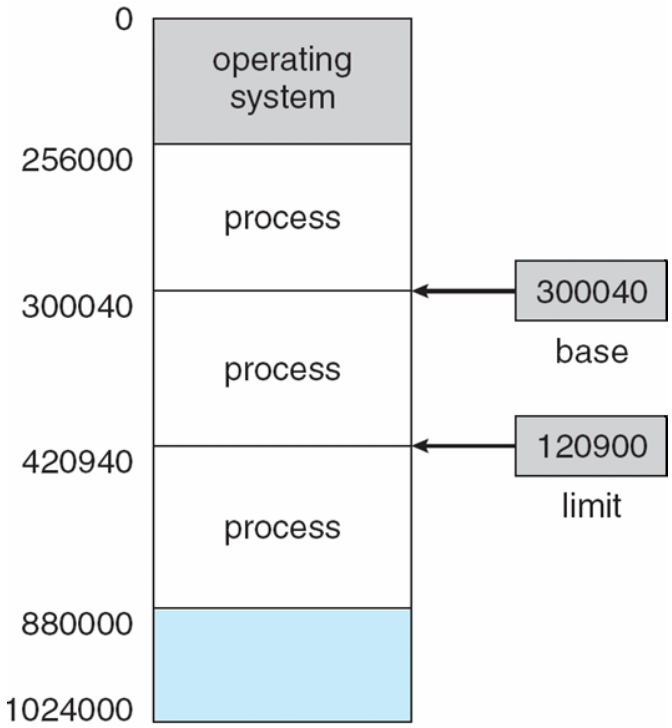
Registres `Base` et `Limit`

Chaque processus reçoit un morceau contigu de mémoire

Le *noyau* contrôle `limit` et `base`

CPU vérifie chaque accès mémoire

Adresses entre `base` et `limit`



Le noyau peut
- accéder à n'importe quelle adresse
- garantir la protection mutuelle des processus

Adresses

Programmes stockés sur disque

Adresses à l'exécution dépendent de `base`

Différentes adresses à différentes étapes d'un programme:

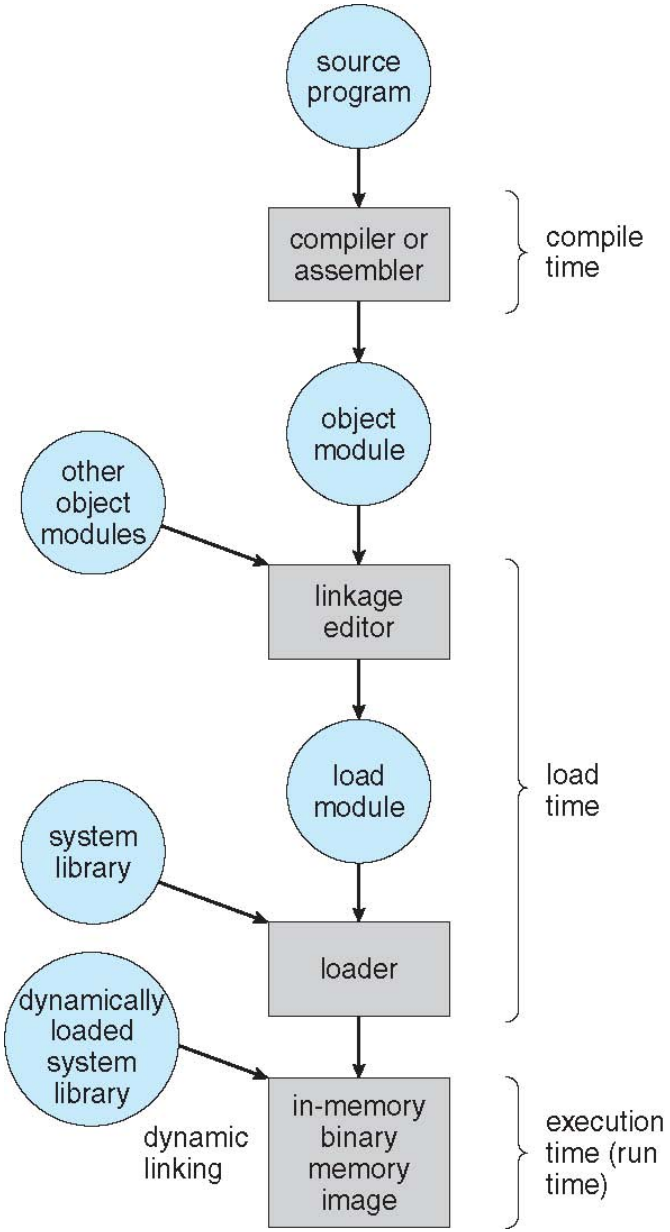
- Code source: adresses généralement symboliques (identificateur)
- Code compilé: adresses *relocatable*

  e.g. "14 bytes après le début de ce module"
- Le *linker* combine des modules en un programme

  e.g. "74014 bytes après le début du programme"
- Le *loader* converti en adresse d'exécution

  e.g. "adresse 75014" (si `base` vaut 1000)

Vie d'un programme



Adresses logiques *vs* physiques

La traduction des adresses ne s'arrête pas avec le *loader*

Les processus fonctionnent avec des adresses *logiques* (aka *virtuelles*)

Le CPU les traduit en adresses *physiques*

Les processus n'ont pas besoin de savoir où ils sont placés

Les processus ne peuvent généralement pas savoir

virtual address space: toutes les adresses *logiques*

physical address space: ensemble des addresses *physiques*

Memory Management Unit (MMU)

Traduire les adresses *logiques* en adresses *physiques*

Souvent plusieurs fois par instruction!

Partie intégrante du CPU

Différentes techniques de traductions

Compromis entre flexibilité et exécution “instantanée”

Inclut généralement *vérification* de droits d’accès

Allocation contiguë

`base` devient un registre de *relocation*

Chaque processus voit un espace *logique* de 0 ... `limit`

Espace *physique* correspondant: `base` ... `base+limit`

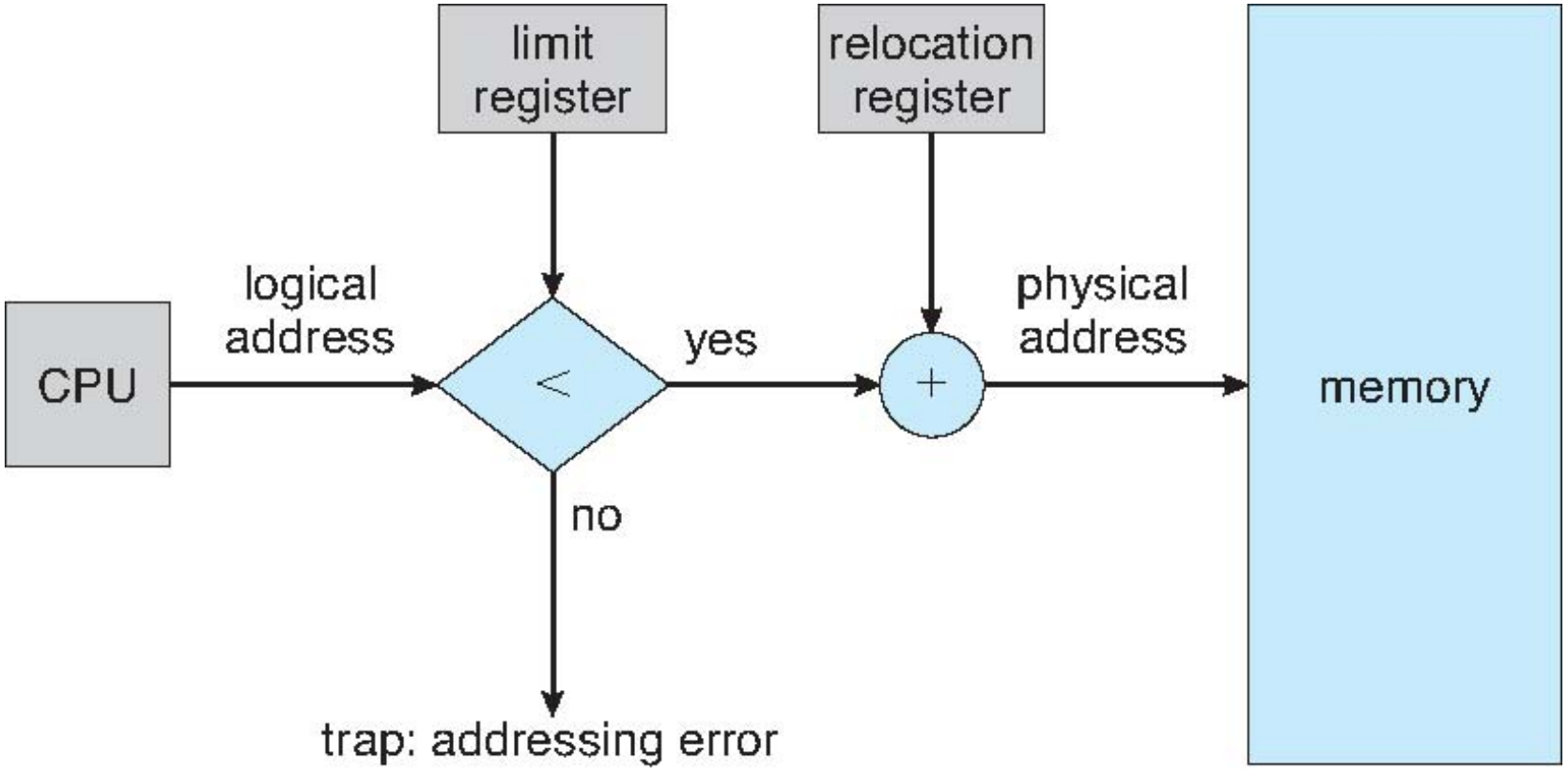
Besoin d'un MMU qui fait cette traduction

Le *noyau* contrôle `base` et `limit`

Le noyau peut placer les processus à sa guise

Le noyau peut déplacer les processus

MMU avec `base+limit`



Gestion mémoire contiguë

Garder trace des zones *physiques* sont utilisées par chaque processus

Garder trace des zones encore libres

Lorsqu'un processus se termine, sa zone est libérée

Au démarrage d'un processus, il faut trouver un "trou" correspondant

Un processus peut demander de la mémoire supplémentaire

Problème de l'allocation mémoire dynamique

Comment satisfaire une requête de N bytes?

- *First fit*: utilise le premier trou assez grand
- *Best fit*: utilise le plus petit trou assez grand
- *Worst fit*: utilise le plus gros trou

Même problème que pour `malloc`

Fragmentation

Fragmentation externe: mémoire inutilisable parce que trop petite

Exemple: il y a assez de mémoire libre,

mais fragmentée en plusieurs trous tous trop petits

Fragmentation interne: mémoire gaspillée par le système

Exemple: le processus a besoin de 600KB mais le SE alloue

par morceaux de 1MB, laissant 400KB inutilisés

Fragmentation totale peut être de l'ordre de 33%

Traduction d'adresses permet heureusement de *compacter*

Segmentation

Généralisation de `base+limit`

Diviser l'espace *logique* en *segments*

Chaque segment a sa propre `base` et `limit`

Adresses logiques de la forme

SegID	*Offset*

Adresse physique = $\texttt{base}[\mathit{SegID}] + \mathit{Offset}$

Vérification de borne: $\mathit{Offset} < \texttt{limit}[\mathit{SegID}]$

Exemples de segments: code, pile, variables globales, bibliothèque, ...

Autopsie de segmentation

La segmentation est d'usage marginal de nos jours

Jamais assez de segments: on aimerait un segment par *objet*

Trop de segments: trop de `base+limit` à garder dans le CPU

Segments pas assez grands: un objet peut occuper toute la mémoire

Adresses trop grandes

Pagination

Diviser l'espace *logique* en *pages*

Diviser l'espace *physique* en *frames*

Chaque *page* et *frame* a la même taille (e.g. 4KB)

Adresses logiques de la forme

PageNb	*Offset*

Adresse physique correspondante:

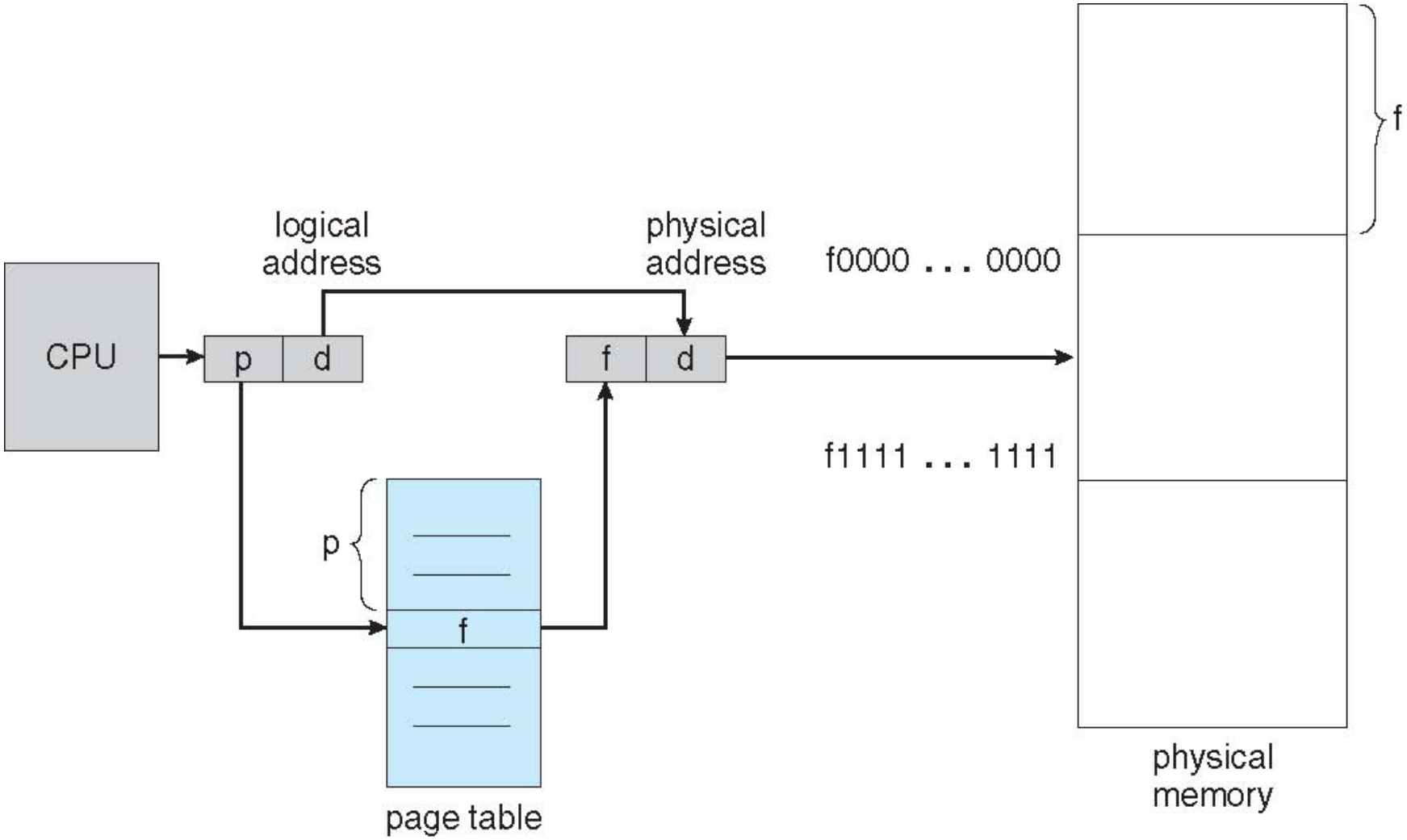
`pagetable`[*PageNb*]	*Offset*

N'importe quelle page peut être dans n'importe quelle frame

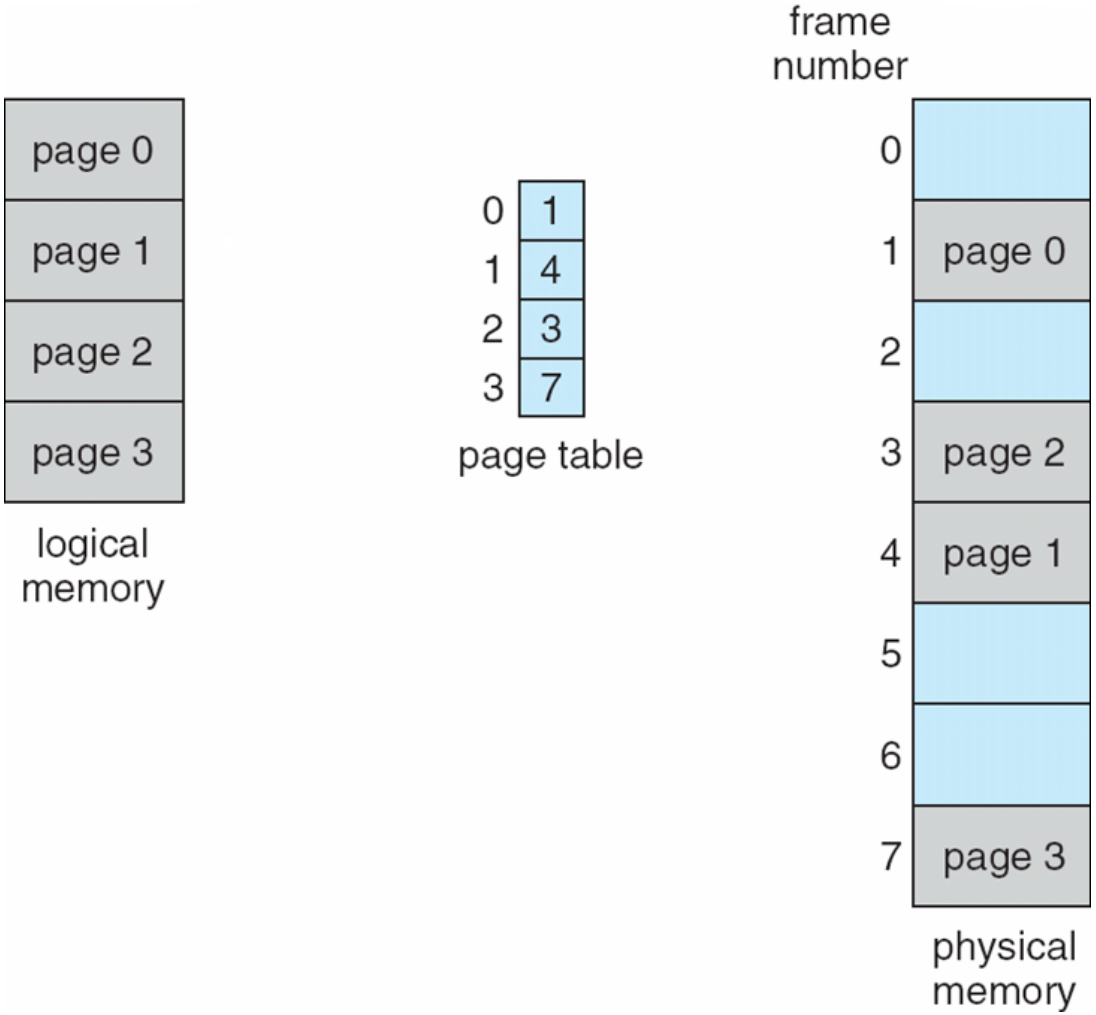
Pas de fragmentation externe

Fragmentation interne à cause de l'allocation par page

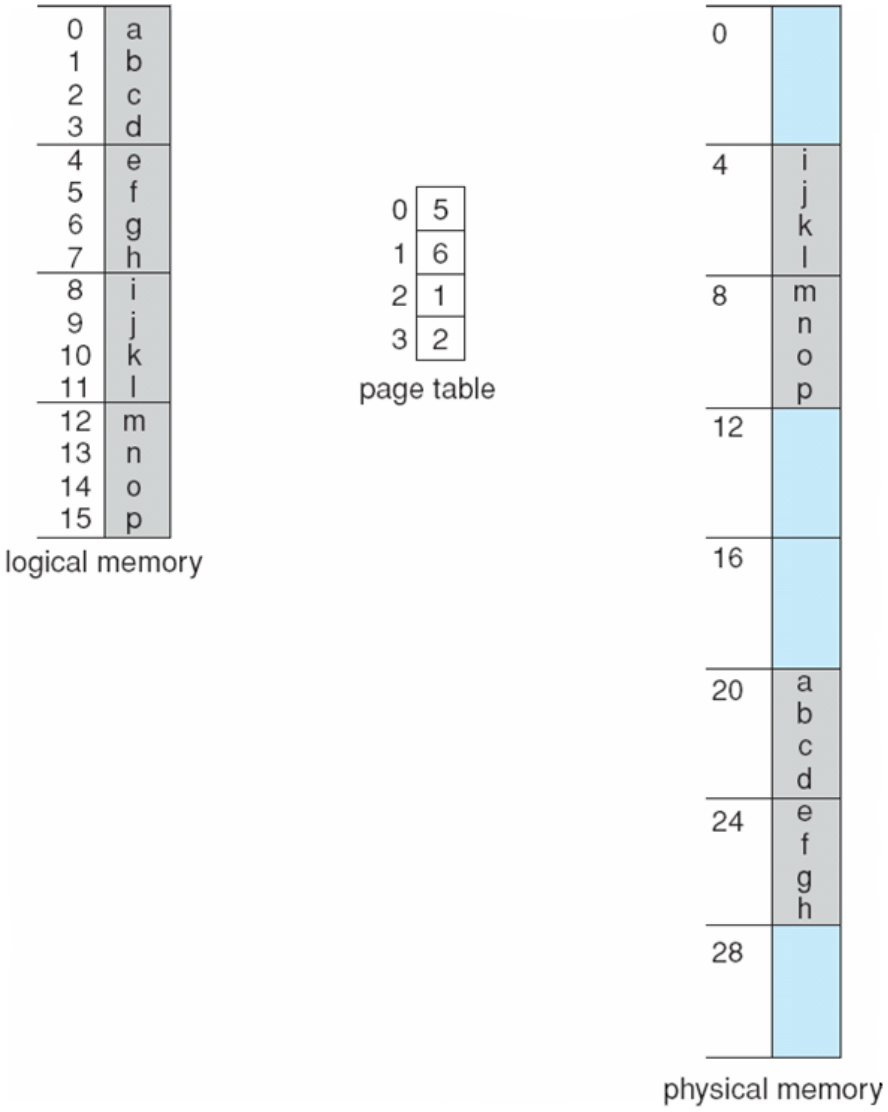
Pagination dans le MMU



Modèle de pagination



Exemple de pagination



Coût de la pagination

Pages plus petites $\Rightarrow$ moins de fragmentation interne

Pages plus petites $\Rightarrow$ plus de pages

Table des page proportionnelle au nombre de pages!

Adresses de 32bit, pages de 4KB $\Rightarrow$ 1M pages (par processus)

Adresses de 64bit, pages de 4KB $\Rightarrow$ 4 milliard de fois plus

Implémentation d'une table de pages

Les tables de pages sont gardées en mémoire

Page table base register donne l'adresse de la table

Page table length register donne la taille de la table

$\Longrightarrow$ **2** accès mémoire par "accès mémoire"!

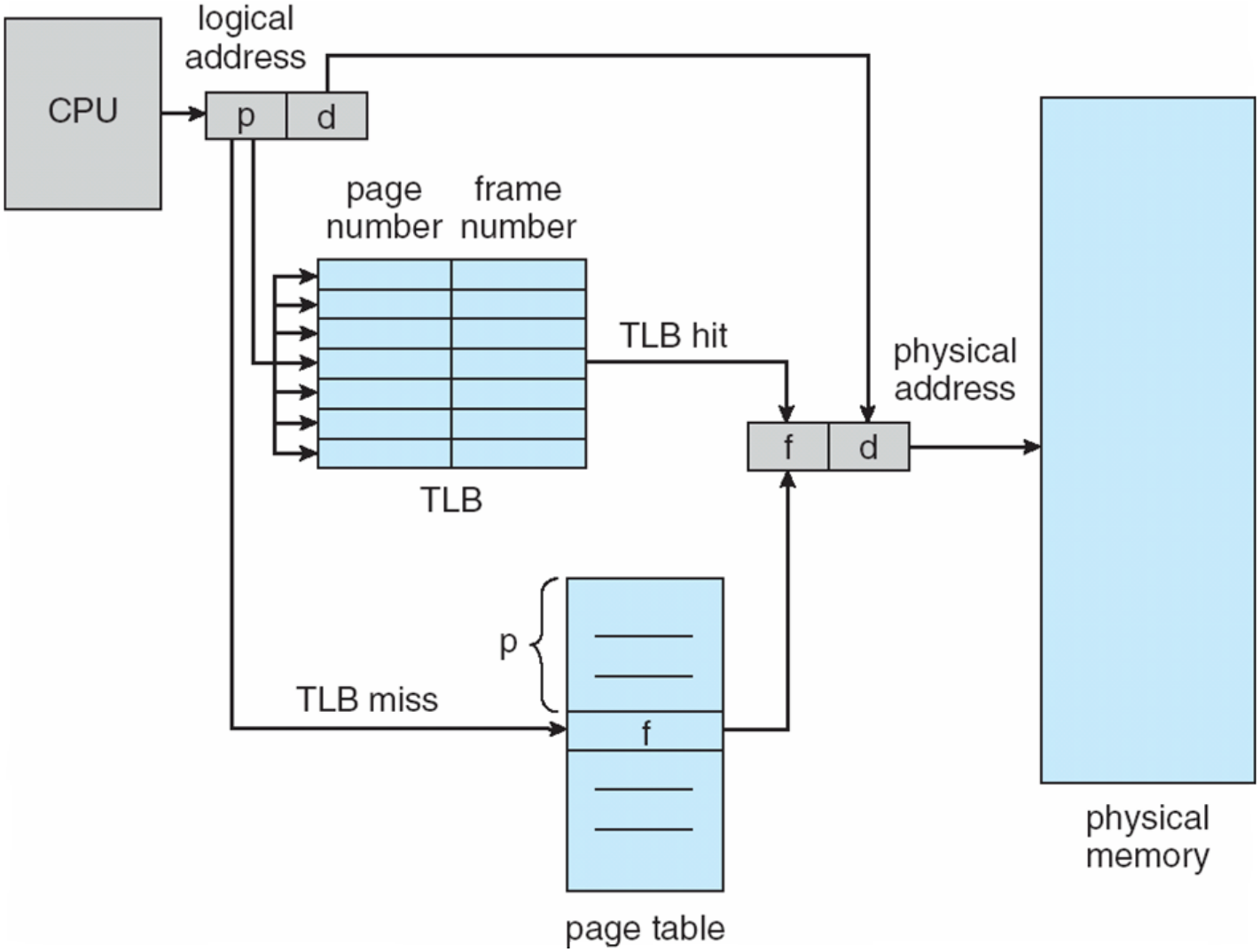
TLB (Translation Look-aside Buffer): cache de la table des pages

TLB est rapide et dans le CPU, mais petit: 64-1024 entrées

Si un numéro de *page* n'est pas dans le TLB:

- Chercher le *frame* correspondant dans la table des pages
- Placer le résultat dans le TLB (remplacement par LRU, FIFO, ...)

Modèle de TLB



Temps d'accès effectif

L: Temps de recherche dans le TLB

α: TLB miss ratio

M: Temps d'accès à la mémoire

$$EAT = (1 - \alpha)(L + M) + \alpha(L + 2M) = L + (1 + \alpha)M$$

Bien sûr M est généralement une formule similaire (cache mémoire)

Protection mémoire

Chaque entrée de la table des pages contient des bits de protection:

- *Valid* bit indique si la page existe vraiment
- *R/W/X* bits pour interdire l'accès en lecture/écriture/exécution

Peut remplacer le *page table length register*

Permet d'utiliser un espace mémoire non-contigu

Toute violation cause un *trap* vers le noyau

Éviter la duplication

La table des pages est une *fonction*

I.e. à chaque *page* de chaque processus correspond au plus une *frame*

Mais par forcément *injective*

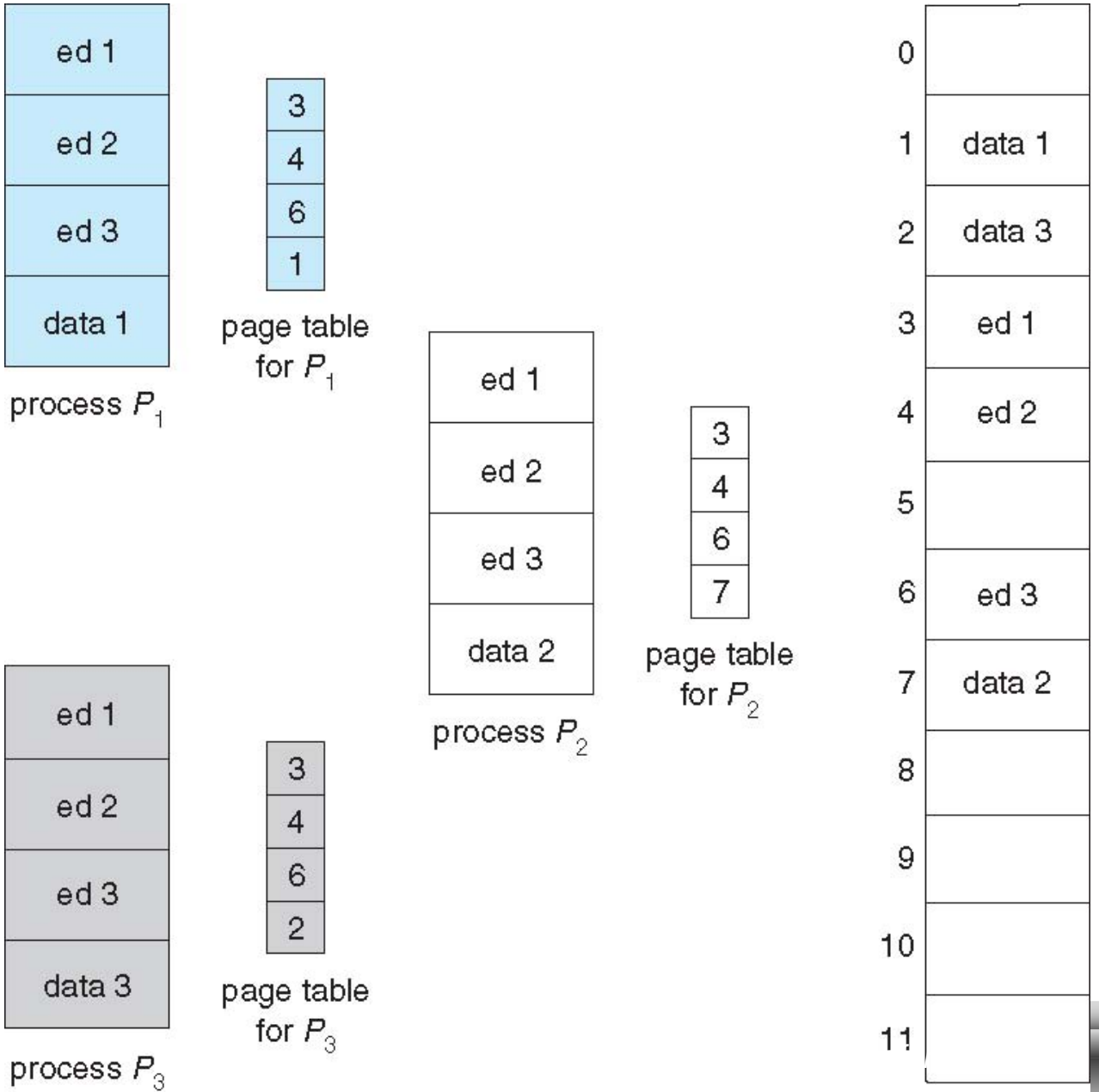
Plusieurs processus peuvent partager les même *frames*

Pas forcément à la même adresse logique, d'ailleurs

- Code d'un programme exécuté plusieurs fois
- Librairie utilisée par plusieurs processus
- POSIX shared memory

Des pages d'un processus peuvent référer aux mêmes frames

Exemple de partage de mémoire



Implémentations de tables de pages

Placer une grosse table en mémoire pose problèmes (fragmentation)

Beaucoup de pages invalides (espace logique non-contigu) coûte cher

$\Longrightarrow$ Tables de pages plus complexes

- Table de pages hiérarchique
- Table de pages inversée
- Table de pages en logiciel
- ...

Table de pages hiérarchique

Au lieu de diviser les adresses en | *PageNb* | *Offset* | avec:

$$\textit{FrameNb} = \texttt{pagetable}[\textit{PageNb}]$$

On divise les adresses en | $\textit{PageNb}_1$ | $\textit{PageNb}_2$ | *Offset* | avec:

$$\textit{FrameNb} = \texttt{pagetable}[\textit{PageNb}_1][\textit{PageNb}_2]$$

Voire | $\textit{PageNb}_1$ | $\cdots$ | $\textit{PageNb}_n$ | *Offset* | avec:

$$\textit{FrameNb} = \texttt{pagetable}[\textit{PageNb}_1]\cdots[\textit{PageNb}_n]$$

Exemple de table de pages à 2 niveaux

un espace d'adressage de 32bit, des pages de 4KB:

$PageNb_1$: 10bit	$PageNb_2$: 10bit	*Offset* : 12bit

`pagetable` est donc un tableau de 2^{10} entrées de 32bit

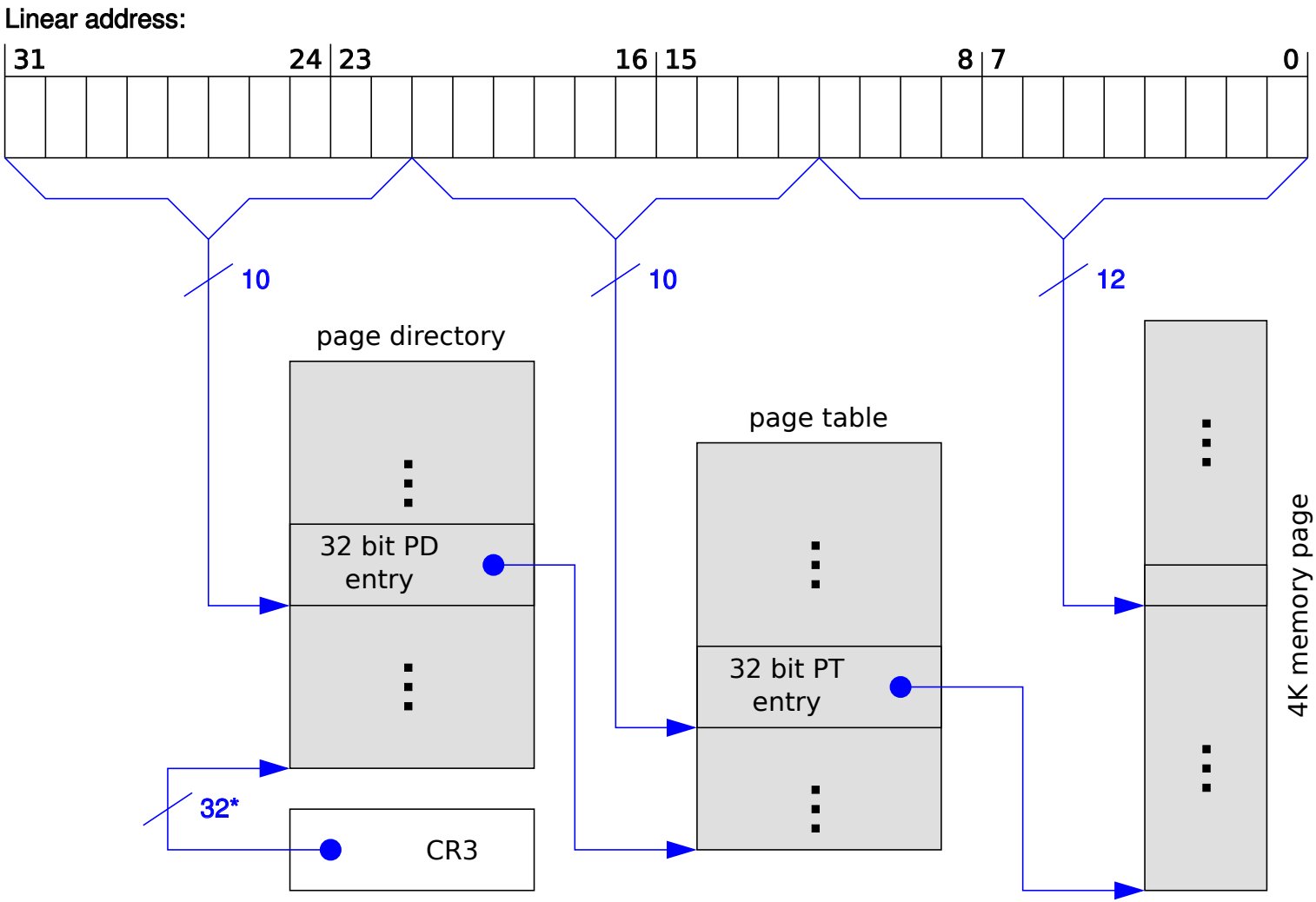
Chaque entrée de ce tableau pointe vers une sous-table

Chaque sous-table est un tableau de 2^{10} entrées de 32bits

Chaque tableau occupe 4KB, donc pas de problème de fragmentation

Note: Sans TLB, il faut 3 "accès mémoire" par "accès mémoire"!

Schéma de table de pages à 2 niveaux



*) 32 bits aligned to a 4-KByte boundary

Table de pages inversée

La table des pages est une grande table de hachage

Taille de la table proportionnelle au nombre de *frames*

Une seule grande table pour tout le système, préallouée

Clé d'accès: *PageNb* et *AddressSpaceNb*

Chaque entrée contient: *FrameNb*, *PageNb*, *ProtBits*, *Next*

Attention au partage des pages

Table de pages virtuelle

Table des pages linéaire “naïve”

La table des pages est elle-même paginée

L’accès `pagetable`[*PageNb*] est fait en adresse *logique*

Donne une structure hiérarchique à plusieurs niveaux

L’arbre est parcouru depuis les feuilles (plutôt que la racine)

Attention au *bootstrap*

Table de pages en logiciel

Nouvelles instructions pour manipuler le TLB

En cas de *miss* dans le TLB, appel au noyau par un *trap*

Le noyau peut alors utiliser n'importe quelle structure de donnée

Peut se simuler avec les bits de validité!