Principes des systèmes d'exploitation

Gestion de la mémoire

UFR IM^{2}AG

M1 MEEF NSI 2022-2023

Renaud Lachaize

Crédits et remerciements

- **Le contenu de ce support de cours est partiellement insipiré, voire emprunté aux travaux d'autres personnes :**

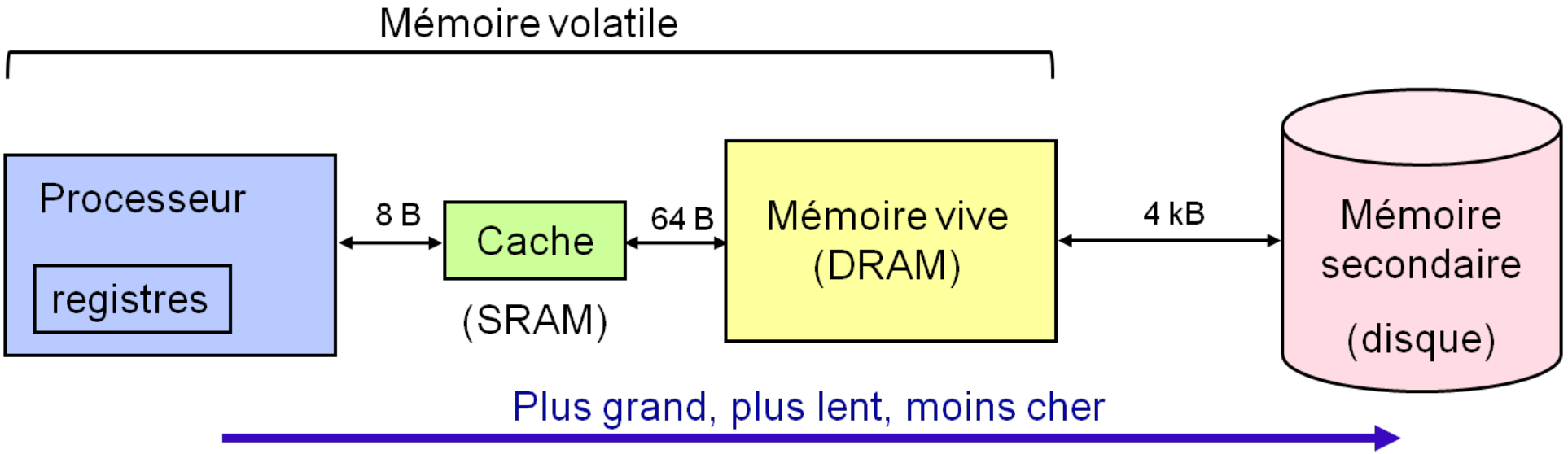
  - Sacha Krakowiak (UFR IM^{2}AG)
  - Randall Bryant, David O'Hallaron, Gregory Kesden, Markus Püschel (Carnegie Mellon University)

  - Ouvrages de référence (voir détails sur la page web) :
    - Silberschatz et al. Operating systems concepts with Java.
    - Tanenbaum. Modern operating systems.
    - Bryant and O'Hallaron. Computer systems: a programmer's perspective.

Plan

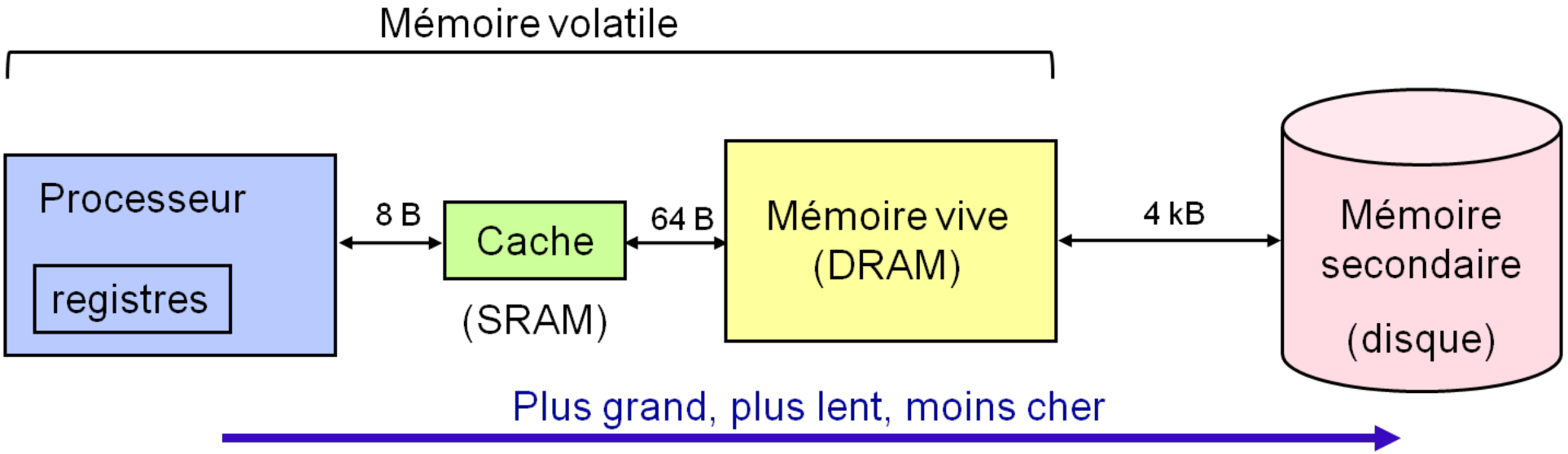
- Hiérarchie mémoire
- Mémoire virtuelle
- Allocation dynamique de mémoire

Hiérarchie mémoire



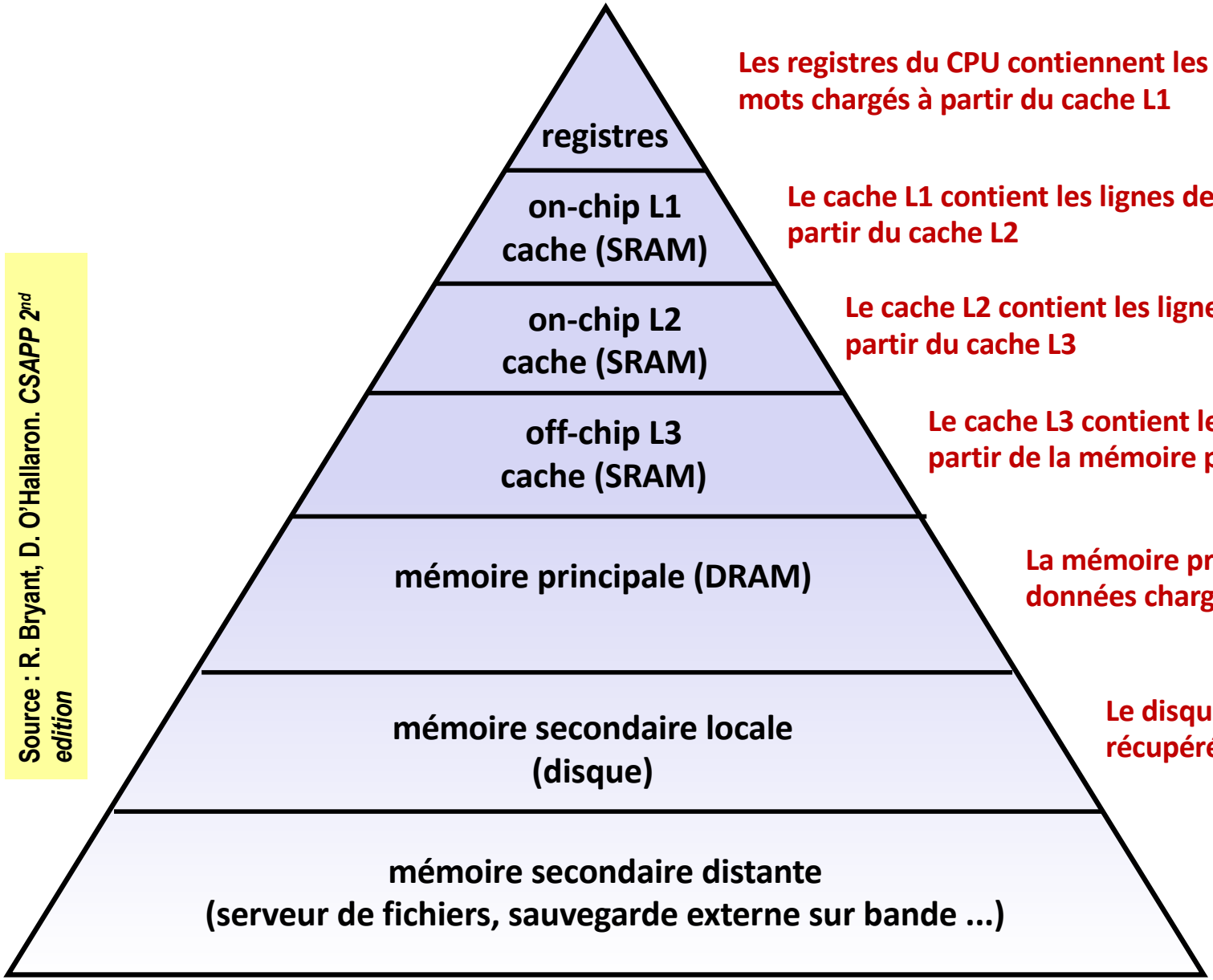
	Registres	**Cache**	**DRAM**	**Disque**
Taille	~ 128 B	~ 32-12 MB	~ 1-8 GB	~ 0,5-2 TB
Temps d'accès	0-1 ns	2-10 ns	40 ns	3 ms
Coût	-	60 $/MB	0,06 $/MB	0,0003 $/MB
Unité de transfert	4-8 B	32-64 B	4-8 kB	

Hiérarchie mémoire



- **L'essentiel des données est conservé en mémoire secondaire**
- **Les données sont chargées en mémoire principale et manipulées dans les registres du CPU**
- **La bonne gestion des transferts entre les différents niveaux de mémoire a un impact primordial sur les performances**

Hiérarchie mémoire
Généralisation du principe



Les registres du CPU contiennent les mots chargés à partir du cache L1

Le cache L1 contient les lignes de cache chargées à partir du cache L2

Le cache L2 contient les lignes de cache chargées à partir du cache L3

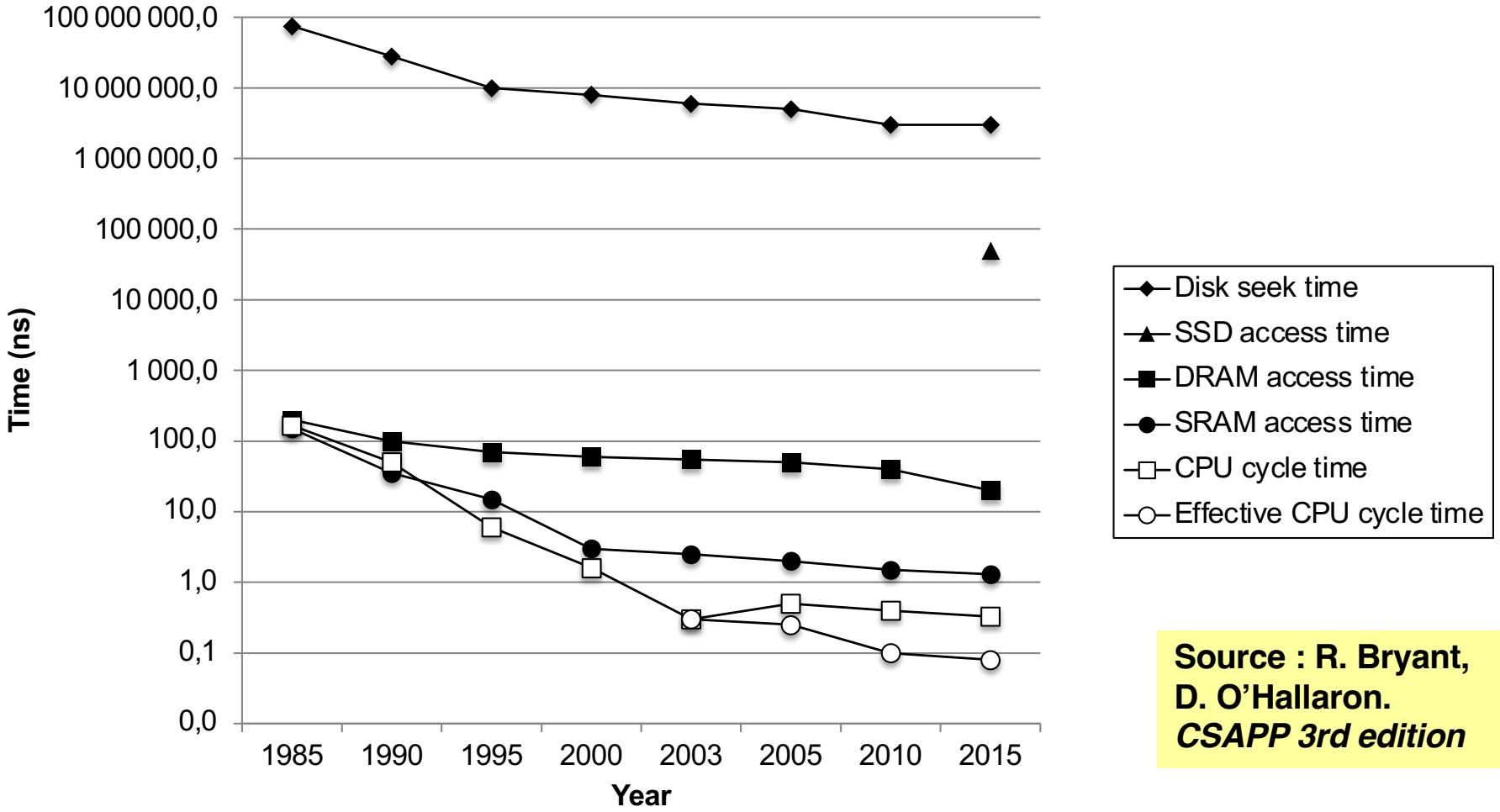
Le cache L3 contient les lignes de cache chargées à partir de la mémoire principale

La mémoire principale contient les blocs de données chargés à partir du disque

Le disque local peut contenir les fichiers récupérés à partir de mémoires distantes

Source : R. Bryant, D. O'Hallaron. *CSAPP 2nd edition*

Hiérarchie mémoire
Tendances quantitatives



Source : R. Bryant, D. O'Hallaron. ***CSAPP 3rd edition***

Notion de « cache »

- **Idée**
  - Améliorer les performances des accès aux données
  - Sans impact majeur sur le coût du matériel
- **Principe**
  - Repose sur la localité temporelle et spatiale des accès à la mémoire
  - Exploiter/intercaler une petite mémoire rapide en amont de la mémoire lente afin de stocker les données les plus importantes au fil de l'exécution d'un programme
  - Principe général, applicable à de nombreux niveaux d'un système
- **Implémentation**
  - Stockage : matériel
  - Gestion : matériel ou logiciel selon les couches considérées
    - consultation, chargement, vidage/arbitrage

Caches : exemples

Type de cache	Quoi ?	Où ?	Latence (cycles)	Géré par
Registres	Mots de 4-8 octets	Au sein du CPU	0	Compilateur
TLB	Informations pour la mémoire virtuelle	À côté du CPU (même puce)	0	Matériel
Cache L1	"Ligne" de 64 octets	Puce	1	Matériel
Cache L2	"Ligne" de 64 octets	Puce ou carte mère	10	Matériel
Mémoire virtuelle	Pages de 4 kB	Mémoire principale	100	Matériel + OS
Cache disque (buffer cache)	Fichiers (complets ou non)	Mémoire principale	100	OS
Cache de fichiers distants	Fichiers (complets ou non)	Disque local	10,000,000	OS (client NFS/AFS ...)
Cache de navigateur web	Pages web	Disque local	10,000,000	Application (navigateur)
Cache web	Pages web	Mémoire ou disque d'un serveur distant	10,000,000 à 1,000,000,000	Application sur serveur

Source : R. Bryant, D. O'Hallaron. *CSAPP 2nd edition*

Hiérarchie mémoire – une analogie

Memory layer	Access latency	Analogy 1	Analogy 2
CPU register	1 cycle ~0.3 ns	1 s	Your brain
L1 cache	0.9 ns	3 s	This room
L2 cache	2.8 ns	9 s	This floor
L3 cache	12.9 ns	43 s	This building
Main memory	120 ns	6 minutes	This campus
Solid state disk (SSD)	50-150 µs	2-6 days	
Hard disk drive (HDD)	1-10 ms	1-12 months	
Main memory of a remote server (over the Internet)	~100 ms	1 century	
Optical storage (DVDs) and tapes	seconds	Several millenia	

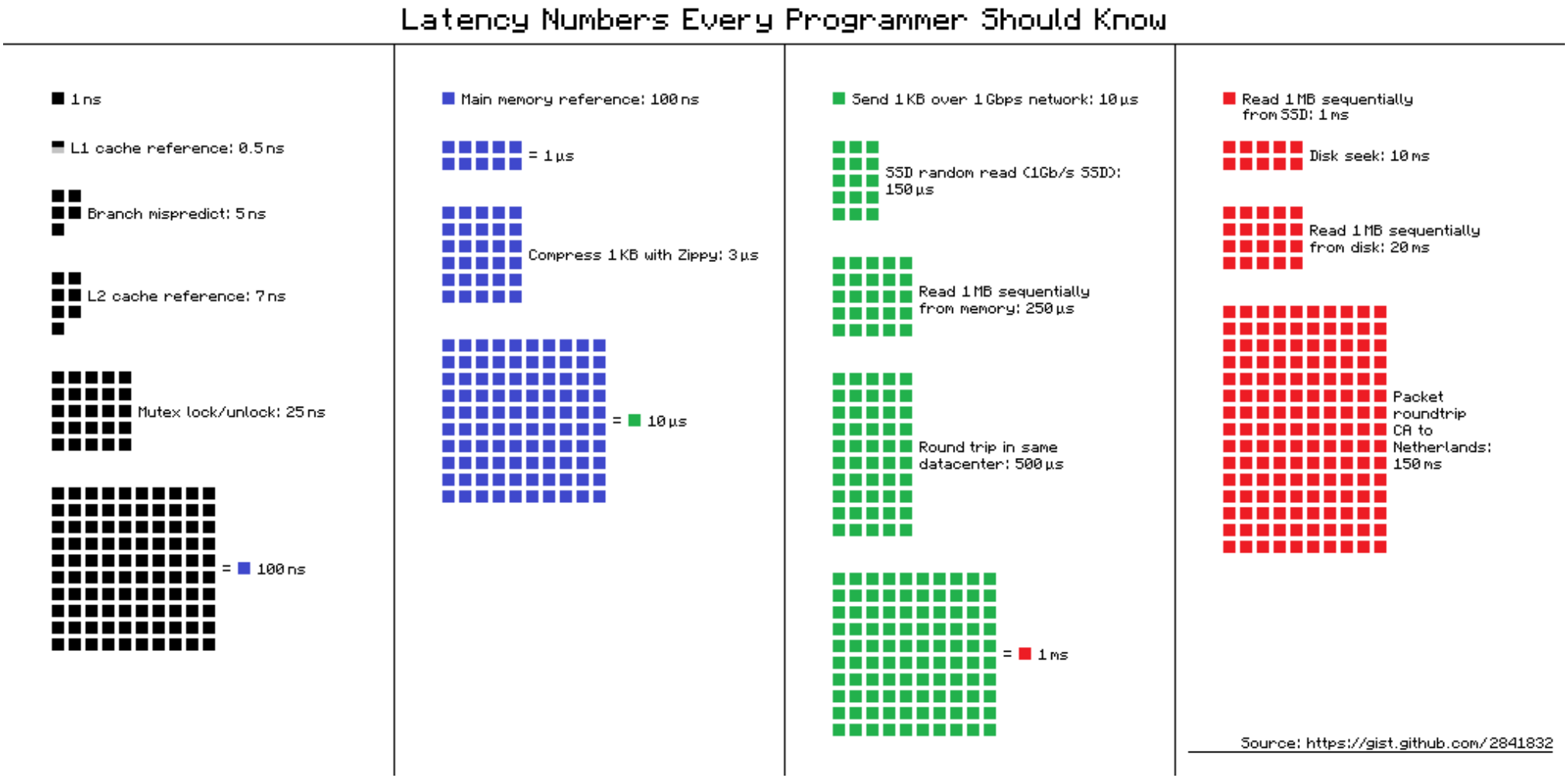
(Inspired by presentations by Jim Gray, Brendan Gregg and Jeff Antwood. http://blog.codinghorror.com/the-infinite-space-between-words/)

Hiérarchie mémoire – Une autre illustration (1/2)

```
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns                      14x L1 cache
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns                      20x L2 cache, 200x L1 cache

Compress 1K bytes with Zippy             3,000   ns        3 us
Send 1K bytes over 1 Gbps network       10,000   ns       10 us
Read 4K randomly from SSD*             150,000   ns      150 us          ~1GB/sec SSD
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us
Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us    1 ms  ~1GB/sec SSD, 4X memory

Disk seek                           10,000,000   ns   10,000 us   10 ms  20x datacenter roundtrip

Read 1 MB sequentially from disk    20,000,000   ns   20,000 us   20 ms  80x memory, 20X SSD
Send packet CA->Netherlands->CA    150,000,000   ns  150,000 us  150 ms
```

- **Sources :**
  - https://gist.github.com/jboner/2841832
  - http://i.imgur.com/k0t1e.png

Hiérarchie mémoire – Une autre illustration (2/2)



Hiérarchie mémoire – Perspectives : NVM (1/2)

- **Une technologie émergente : la mémoire non volatile**
  - *Non-Volatile Memory (NVM)*
  - Parfois aussi appelée *"Storage Class Memory"* (SCM)

- **Comme pour la mémoire vive (RAM) "classique" :**
  - Rapide
  - Directement accessible par le CPU, à la granularité de l'octet

- **Comme pour les disques "classiques":**
  - Haute densité de stockage, faible coût par octet
  - Pas de consommation énergétique en l'absence de lecture/écriture
  - **Persistance des données**

- **Quel impact à prévoir ?**
  - Sur la hiérarchie mémoire
  - Sur les couches logicielles

Hiérarchie mémoire – Perspectives : NVM (2/2)

NVM: Plusieurs technologies physiques

Technology	Read latency	Write latency	Density	Cost
DRAM (baseline)	15 ns	15 ns	Low	$$$$
PCM	50 ns	500 ns	Medium	$$
ReRAM	10 ns	50 ns	High	$$$$
STT-MRAM	10 ns	50 ns	Low	$$$
CNT	< 50 ns	< 50 ns	High	$$$

(source: M. Seltzer et al. An NVM Carol. ICDE 2018.)

Plan

- Hiérarchie mémoire
- Mémoire virtuelle
- Allocation dynamique de mémoire

Rappels : mémoire physique

- **Un processeur doté d'un bus d'adresses de n bits peut désigner 2^n cases mémoire**

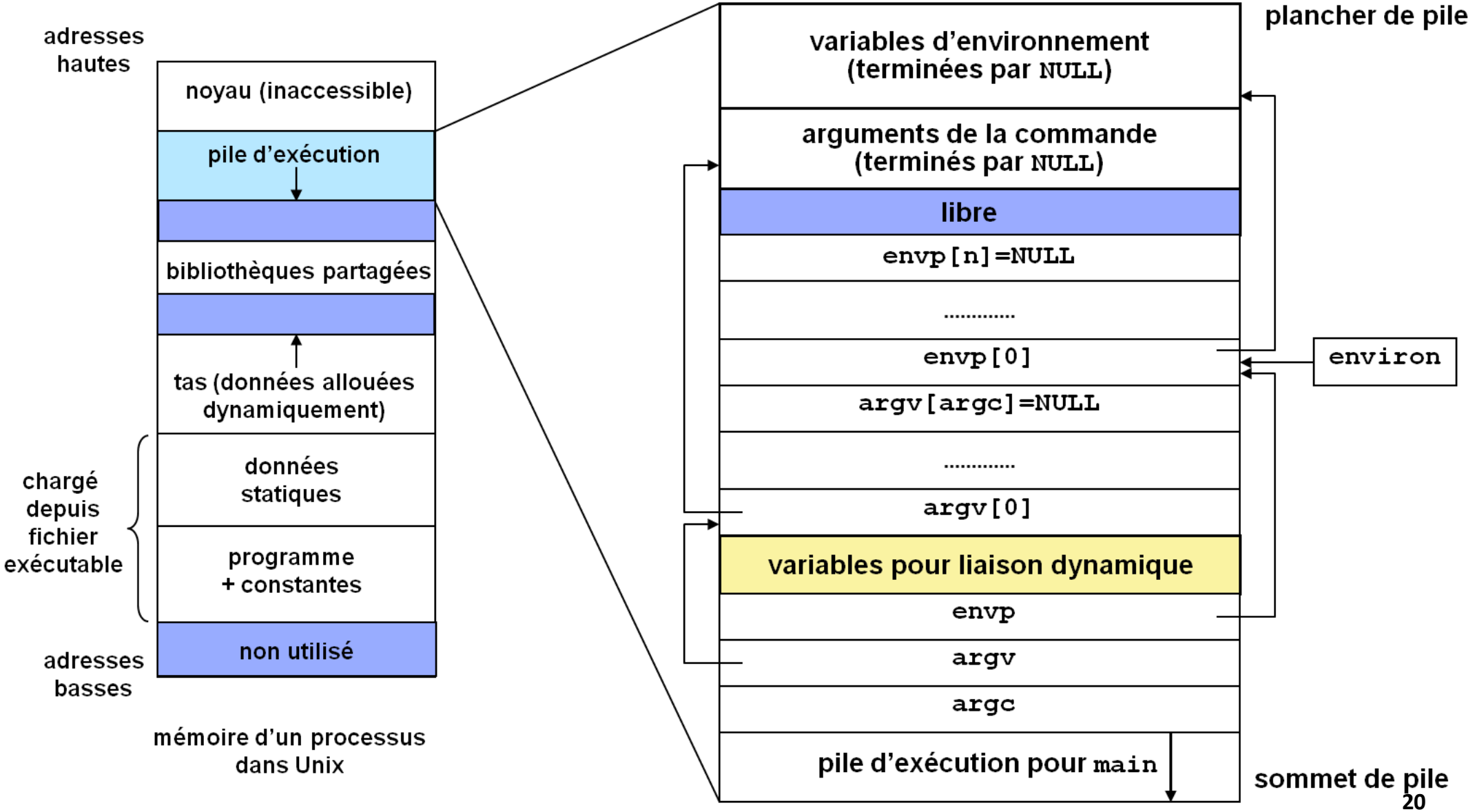
- **Sur la carte mère, un circuit de décodage d'adresses sélectionne le ou les boitiers mémoire à activer, en fonction de :**
  - l'adresse émise sur le bus d'adresses
  - la taille des données et le type d'accès (lecture/écriture)

- **Une adresse de case mémoire peut correspondre à :**
  - un élément de mémoire persistante (ROM) – Exemple : BIOS
  - un élément de mémoire principale volatile (DRAM)
  - un registre d'un coupleur d'E/S
  - aucun élément de mémoire (un accès déclenche un déroutement d'erreur)

Espace mémoire : Le point de vue d'un processus

- **Pour lancer l'exécution d'un processus, on doit placer en mémoire les informations suivantes :**
  - Le code du programme à exécuter
  - Les arguments du programme à exécuter
  - Les variables d'environnement
  - Le code et les données des bibliothèques utilisées par le programme
  - Les valeurs constantes utilisées par le programme
  - Les variables globales du programme
  - Le code et les structures de données du noyau (cf. appels système)
  - Une réserve de mémoire pour :
    - Les variables locales utilisées par les procédures/fonctions
    - La chaîne courante d'appels en cascade de procédures/fonctions
    - Les allocations dynamiques

Espace mémoire d'un processus : vue logique



Mémoire virtuelle : motivations

- **Le concept de mémoire virtuelle a été introduit pour répondre à un grand nombre de problèmes**
  - posés aux programmeurs d'applications, aux développeurs d'outils et aux concepteurs de systèmes
  - essentiellement liés à la simplicité, la souplesse et la sécurité de la gestion mémoire
- **Nous allons passer en revue un sous-ensemble de ces problèmes**

Problème n°1 : capacité mémoire

- **Un seul processus**
  - Comment faire si la taille des informations à manipuler excède la taille de la mémoire principale disponible sur la machine ?
    - Abandon ?
    - Gestion explicite des échanges disque/mémoire dans l'application ?

- **Plusieurs processus**
  - Comment faire si la taille des informations à manipuler pour l'ensemble des processus dépasse la taille de la mémoire principale disponible sur la machine ?
    - Renoncer au (pseudo-)parallélisme ?

Problème n°2 : cohabitation entre processus

- **Relogement**
  - Au chargement du processus i, les cases de mémoire qu'il comptait utiliser son déjà occupées par un autre processus j
  - Besoin de décaler les adresses des informations du processus i

- **Mémoire physique fragmentée**
  - Au fil des lancements et des terminaisons de processus, la mémoire physique disponible peut devenir fragmentée
  - Mais la structure mémoire d'un processus nécessite des plages de mémoire contiguës
  - Que faire ?
    - Renoncer à lancer le nouveau processus ?
    - Tuer un processus existant pour libérer de la place ?
    - Décaler les plages de mémoire utilisées par un processus existant ? (comment ?)

Problème n°3 : contrôle du partage entre processus

- **Isolation**
  - Comment éviter qu'un processus exécutant du code bogué (ou malveillant) ne perturbe (ou espionne) l'exécution d'un autre processus ?
- **Partage**
  - Comment permettre à des processus de mettre en commun des informations pour interagir ?
    - Exemple 1 : code et données du noyau, communs à tous les processus
    - Exemple 2 : tampon circulaire pour prod-cons
- **Éviter le stockage d'informations redondantes**
  - But : gain de place en mémoire centrale
    - Exemple 1 : code commun à plusieurs processus indépendants
    - Exemple 2 : à l'issue d'un fork(), contenus quasi-identiques pour l'espace mémoire du père et celui du fils

Mémoire virtuelle : motivation
Résumé des problèmes de départ

- **L'ensemble des informations nécessaires pour l'exécution d'un processus (code, données) peut dépasser la capacité de la mémoire centrale**
  - Solution rudimentaire : gestion explicite des échanges entre mémoire et disque
  - Cette solution est très restrictive

- **Cohabitation de plusieurs processus en mémoire**
  - Le problème précédent demeure avec plusieurs processus
  - Variante : pas assez d'emplacements contigus pour un processus
  - Nécessité de charger les processus à des adresses différentes

- **Indépendance et protection mutuelle des processus**
  - Nécessité de protéger chaque processus contre les erreurs ou les malveillances des autres
  - Néanmoins, besoin de pouvoir définir des informations communes (noyau, bibliothèques, données)
    - Partage d'informations et gains de place

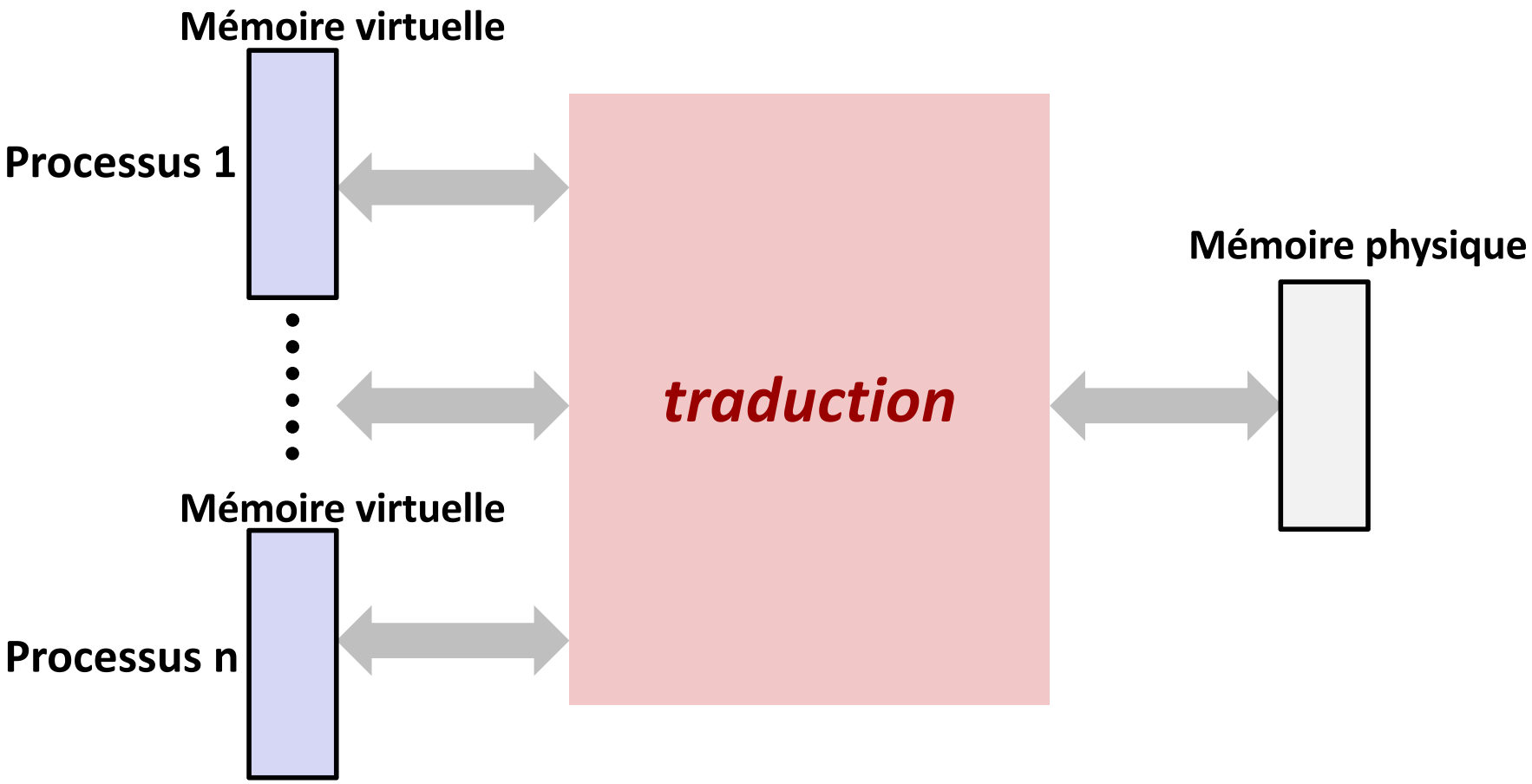
Mémoire virtuelle : principes de la solution (1)

- **Ajout d'un niveau d'indirection pour la désignation mémoire**
  - Un processus manipule exclusivement des adresses virtuelles
    - 2^n adresses virtuelles (0 à 2^{n-1})
    - 2^m adresses physiques (0 à 2^{m-1}), $m \leq n$
    - Un octet d'information à 1 adresse physique et 1 (ou plusieurs) adresse(s) virtuelle(s)
  - Traduction entre adresse virtuelle et adresse physique
    - Effectuée par le « système » (collaboration matériel /noyau)
    - La traduction est contextuelle (dépend de la table de correspondance courante)
  - Intérêts :
    - Évitement des problèmes de placement (adresses identiques, manque de contiguïté)
    - Contrôle de l'accès aux informations
    - Évitement du stockage d'informations redondantes

Mémoire virtuelle : principes de la solution (2)

- **Espace mémoire d'un processus**
  - Chaque processus est associé à un contexte d'adressage distinct
    - Table de correspondance virtuel/physique
  - Au niveau logique, chaque processus peut donc désigner 2^n cases de mémoire
    - Indépendamment des besoins des autres processus
  - Intérêts :
    - Complémentarité avec le point précédent
    - Souplesse et simplicité pour l'organisation interne d'un processus

Mémoire virtuelle : principes de la solution (3)



Source : R. Bryant, D. O'Hallaron. *CSAPP 2nd edition*

Mémoire virtuelle : principes de la solution (4)

- **Gestion de la place disponible en mémoire physique**
  - À un instant donné, un processus n'a besoin que d'un sous ensemble de ses informations (une partie de son code, de sa pile, de ses variables ...)
  - Idée :
    - Ne stocker que ces informations en mémoire centrale
    - Le reste peut être stocké dans la mémoire secondaire
  - La mémoire centrale est vue comme un « cache » des informations disponibles en mémoire secondaire
  - Le système (matériel+OS) détecte quelles sont les informations nécessaires (ou pas) et réalise les transferts correspondants entre la mémoire principale et la mémoire secondaire

  - Intérêt : indépendance (au niveau logique) d'un programme par rapport à la place disponible en mémoire centrale

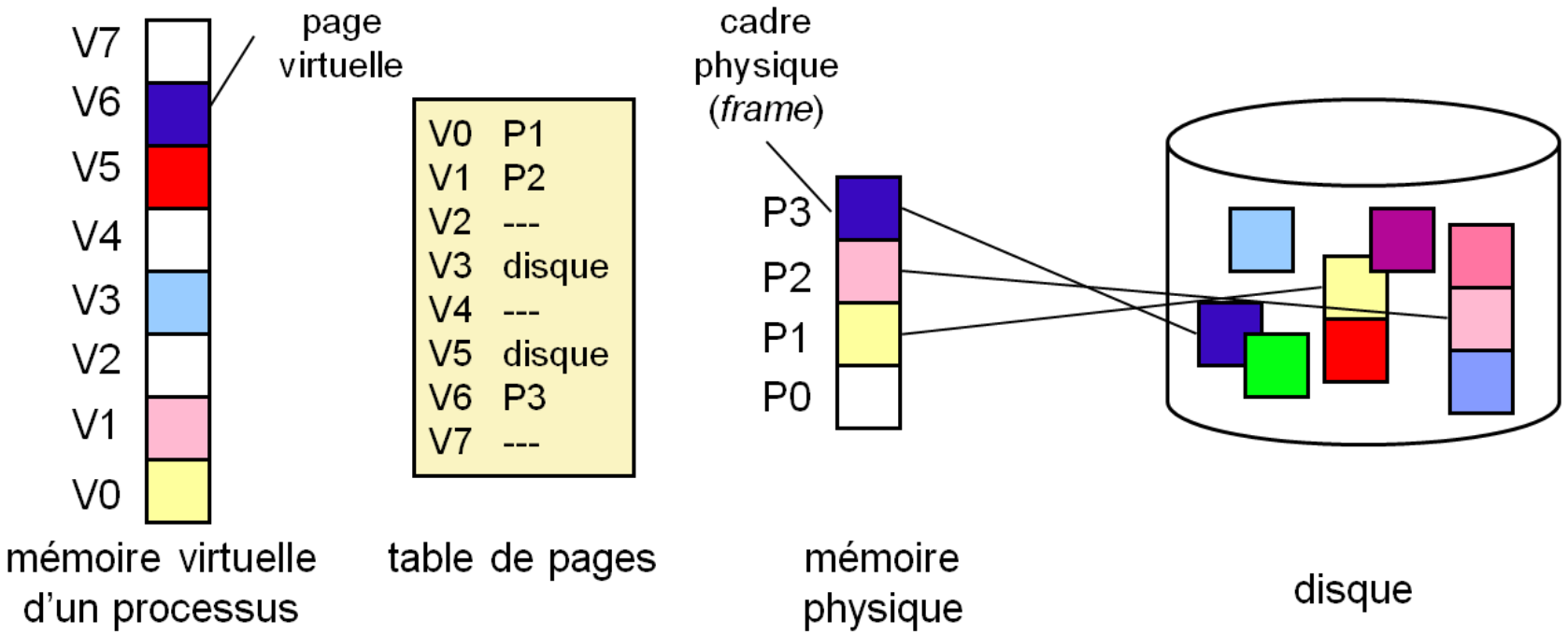
Principe de la mémoire virtuelle paginée (1)

- **La mémoire principale et le disque sont organisés comme un ensemble de pages de taille fixe**
  - Typiquement de l'ordre de 4 à 8 kilo-octets.
  - Si on veut différencier contenant et contenu on parle respectivement de

cadres (*frames*) et de pages

- **Pourquoi une taille fixe ?**
  - Commodité de gestion (les cadres sont interchangeables)
  - Efficacité (taille des structures de données, amortissement du coût)
- **À un instant donné,**
  - la mémoire physique ne peut contenir qu'un (petit) sous-ensemble des pages constituant l'ensemble des mémoires virtuelles
  - mais les autres pages sont toutes présentes en mémoire secondaire

Principe de la mémoire virtuelle paginée (2)



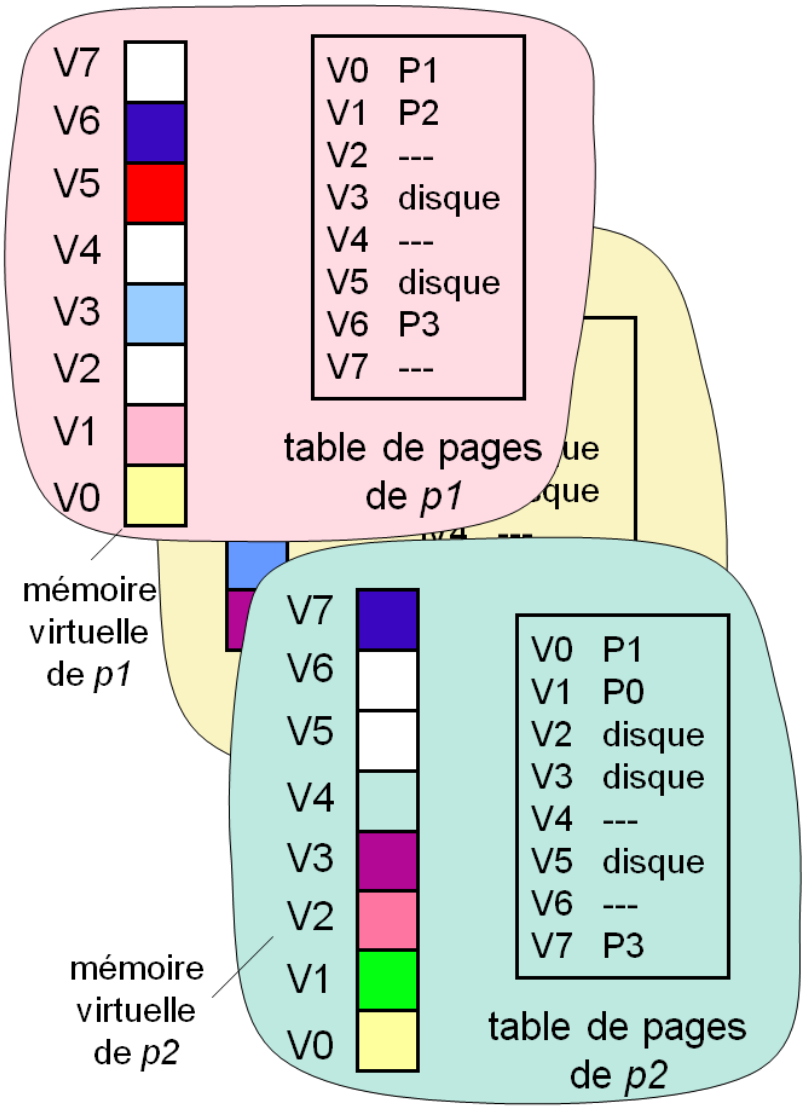
La mémoire virtuelle peut être plus grande que la mémoire physique (mais pas plus grande que la capacité d'adressage du processeur).

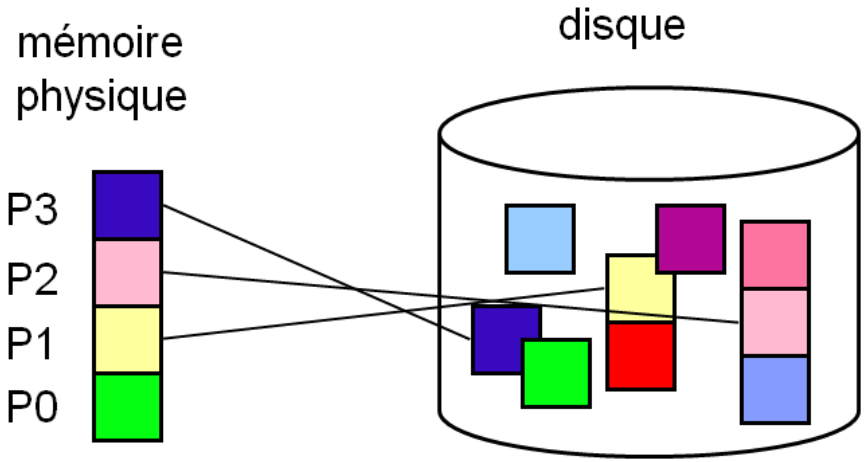
Une page virtuelle dont le contenu est en mémoire physique est immédiatement accessible.

Une page virtuelle dont le contenu n'est que sur disque doit être amenée en mémoire physique pour être accessible ; il faut pour cela lui trouver un cadre libre.
S'il n'y a plus de cadre libre, il faut en libérer un (donc copier son contenu sur disque s'il a changé).

Principe de la mémoire virtuelle paginée (3)

Cas de plusieurs processus





Chaque processus possède sa propre mémoire virtuelle, indépendante de celle des autres processus (il a donc sa propre table de pages).
Des processus peuvent partager des pages, à la même adresse (page V0) ou à des adresses différentes (page V6 de *p1* et V7 de *p2*)

Pagination à la demande

Lorsque le processeur accède à une adresse virtuelle dans une page V, il y a 3 cas, selon l'état de V dans la table des pages (nous ne considérons pas la protection, cf. plus loin)

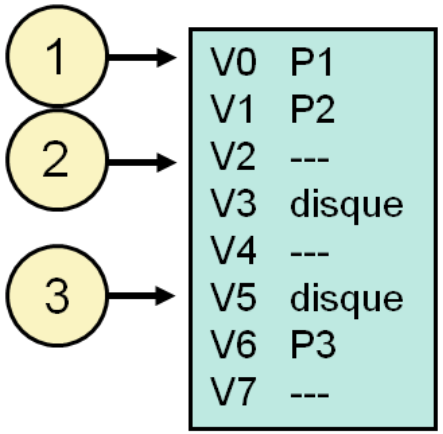


Table de pages

Cas 1 (exemple : V0) : la page correspondante est en mémoire physique. L'accès est immédiat.

Cas 2 (exemple : V2) : il n'y a pas de contenu associé à V : erreur de segmentation (vient d'une faute de programmation).

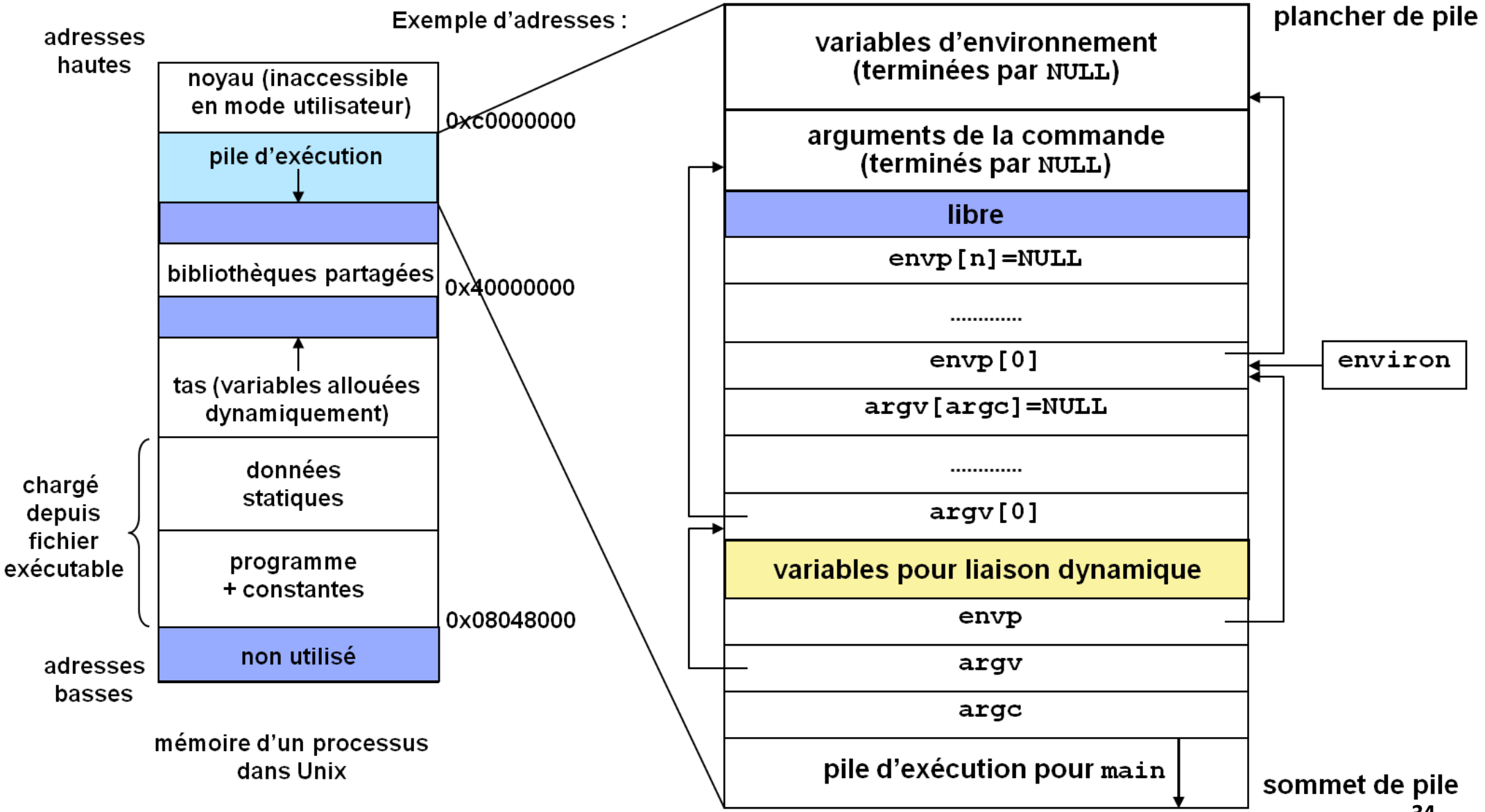
Cas 3 (exemple : V5) : la page correspondante est sur disque (à une adresse indiquée). Il y a défaut de page.

Traitement du défaut de page

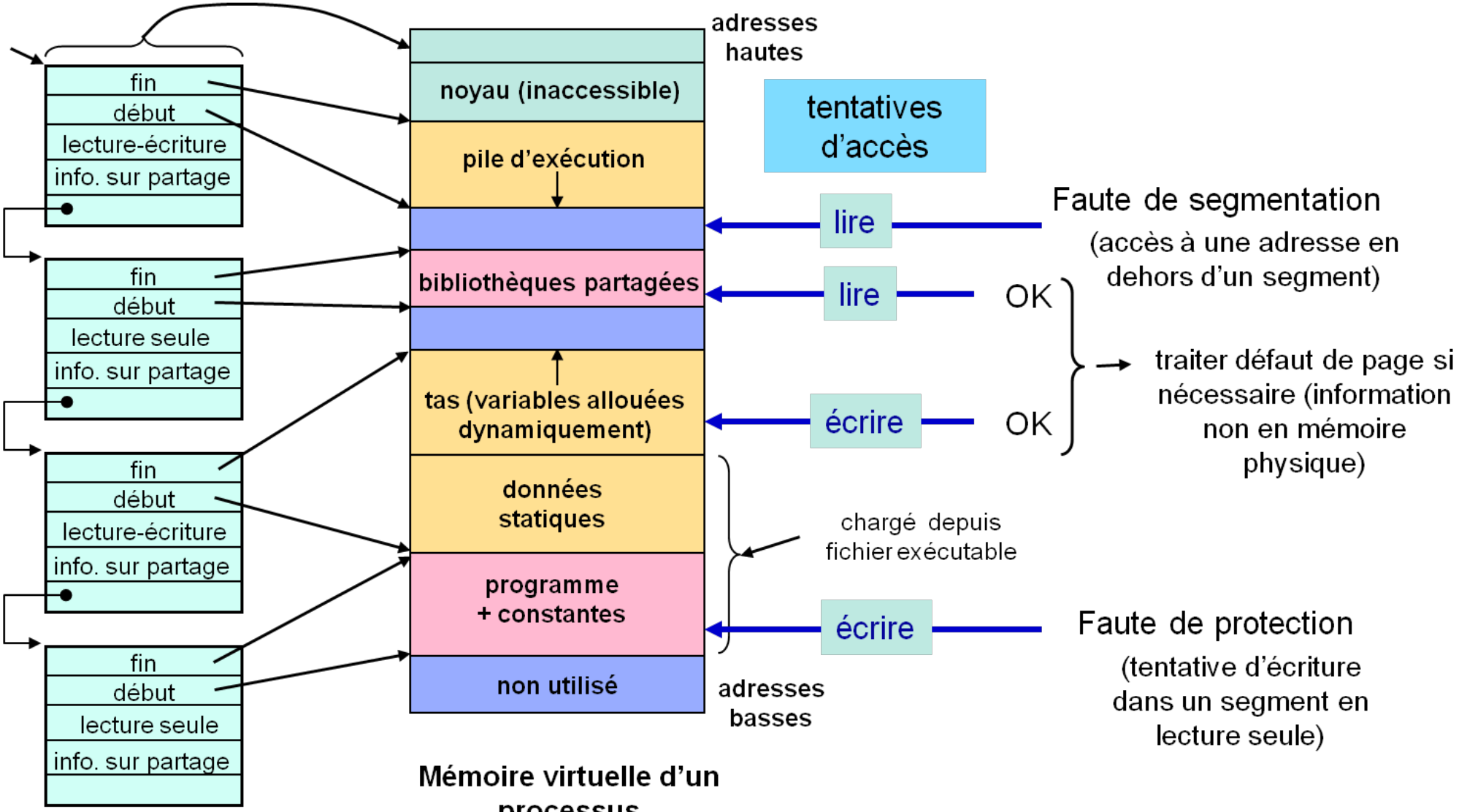
Le processus demandeur est bloqué (le processeur est donc alloué à un autre processus). Le système d'exploitation trouve un cadre libre en mémoire physique et y charge la page depuis le disque. Quand la page est chargée, la table des pages est mise à jour et le processus est réveillé.

Les défauts de page sont très coûteux. Il faut donc réduire leur nombre.

Organisation et protection de la mémoire virtuelle (1)



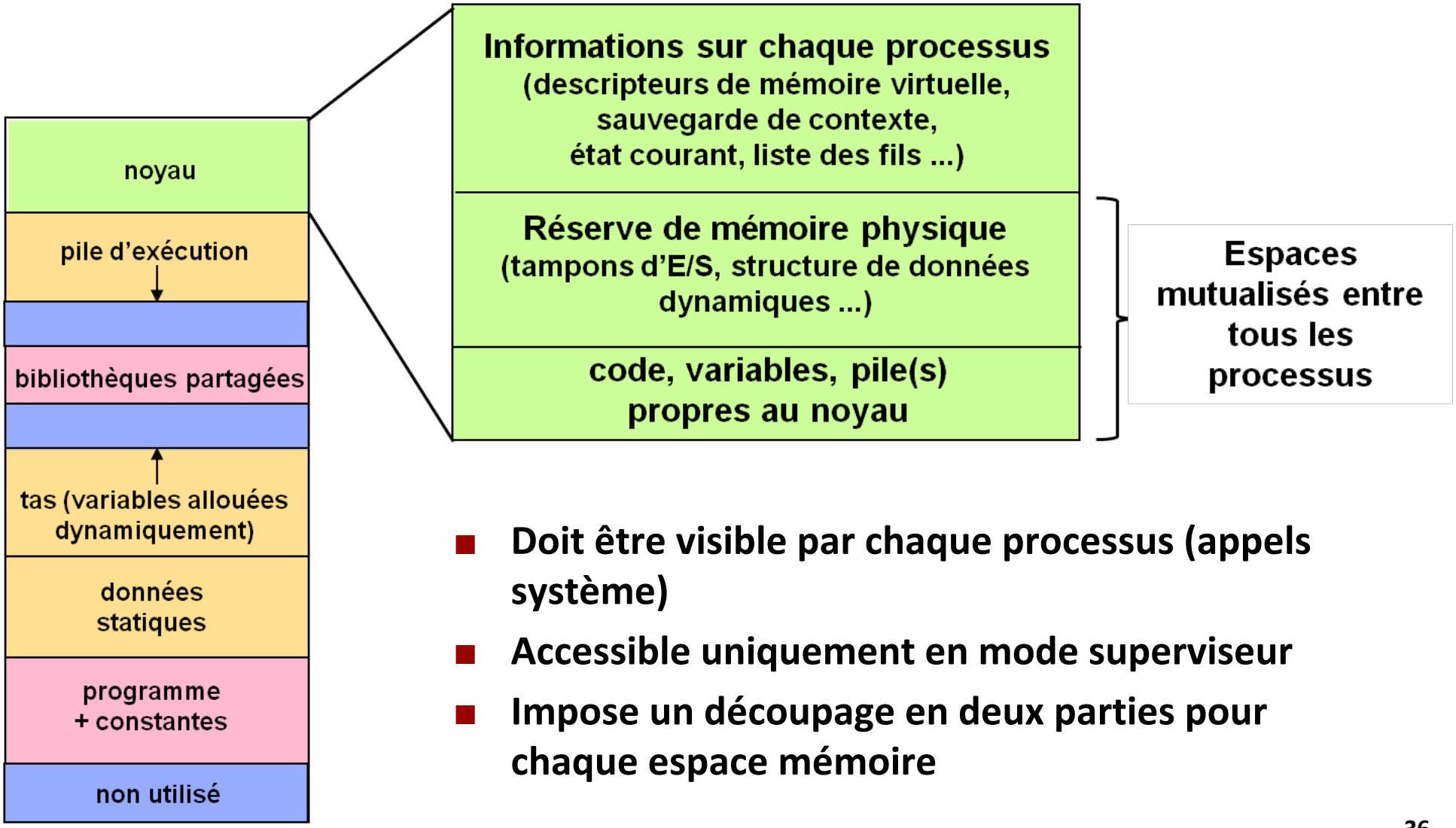
Organisation et protection de la mémoire virtuelle (2)



Mémoire virtuelle d'un processus dans Unix

Structures pour un processus

Organisation et protection de la mémoire virtuelle (3)
Le noyau du système d'exploitation

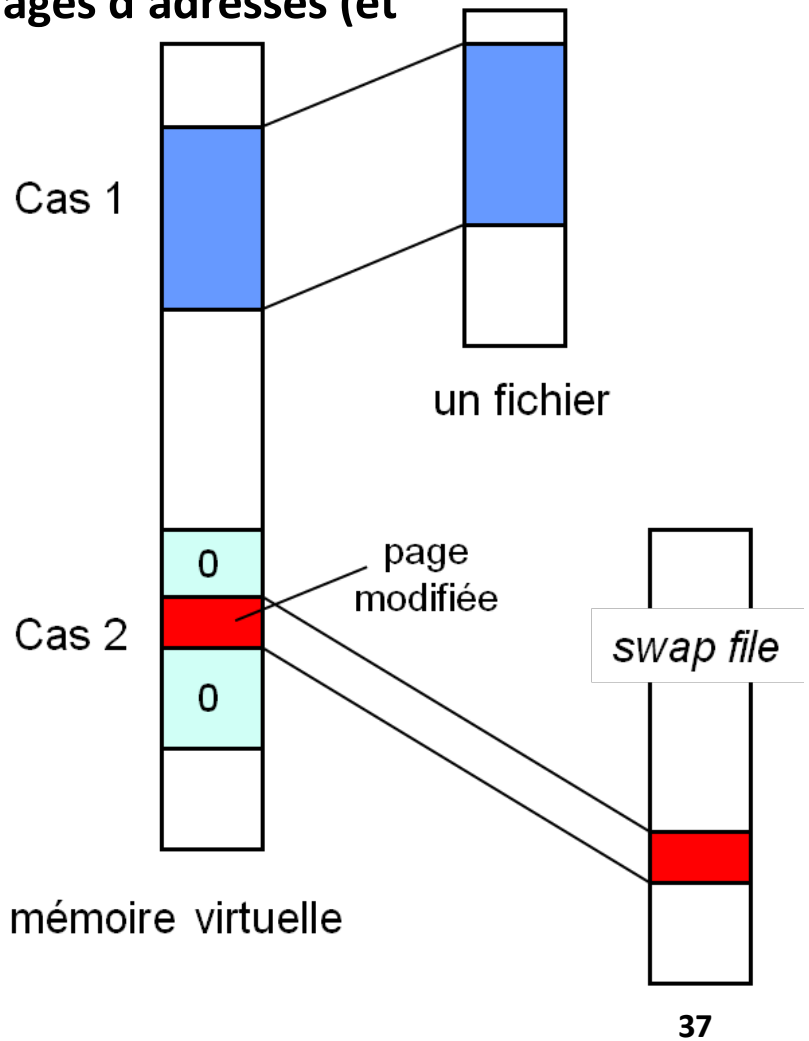


- Doit être visible par chaque processus (appels système)
- Accessible uniquement en mode superviseur
- Impose un découpage en deux parties pour chaque espace mémoire

Construire une mémoire virtuelle

- Le contenu d'une mémoire virtuelle est défini par les différentes zones (ou segments) qui sont associés (ou couplés) à ses diverses plages d'adresses (et spécifiés par des descripteurs).
- Un segment peut être de deux natures.

1. Associé à un fichier (ou à une région contiguë d'un fichier). Il a donc une image persistante sur disque. Néanmoins tant qu'il n'y a pas accès effectif à une page de la zone couplée, il n'y a pas transfert d'information depuis le disque (on rappelle que la pagination est à la demande) Exemple : code.
2. Associé à un fichier "fictif" rempli de zéros. Au premier accès à une page, celle-ci est mise à zéro, sans transfert depuis le disque. Un tel segment est dit "0-demande" puisqu'il n'y a pas initialement de fichier qui lui soit associé. Si une page d'un tel segment est modifiée, elle est associée à un fichier spécial dit *fichier d'échange (swap file)*.
   Exemples : pile, tas.



Fichier d'échange

- **Fichier spécial sur disque (ou partition) nécessaire pour gérer la mémoire virtuelle (capacité supérieure à celle de la mémoire physique)**
  - En anglais : *swap file / swap partition*
  - Utilisé pour stocker les pages qui ne sont pas activement utilisées (à un instant t) et qui contiennent des données *privées* (pile, tas, variables globales) à chaque processus
  - Remarque : les pages qui contiennent les informations telles que le code des programmes, ou données communes à plusieurs processus (fichiers de données) n'ont pas à être stockés dans le fichier d' échange
  - La taille du fichier d'échange (sur disque) + la taille de la mémoire physique bornent la quantité totale de mémoire virtuelle que le système peut allouer
  - Le fichier d'échange est stocké sur disque (il est donc persistant) mais son contenu est réinitialisé à chaque redémarrage du système

Support matériel pour la mémoire virtuelle (1)

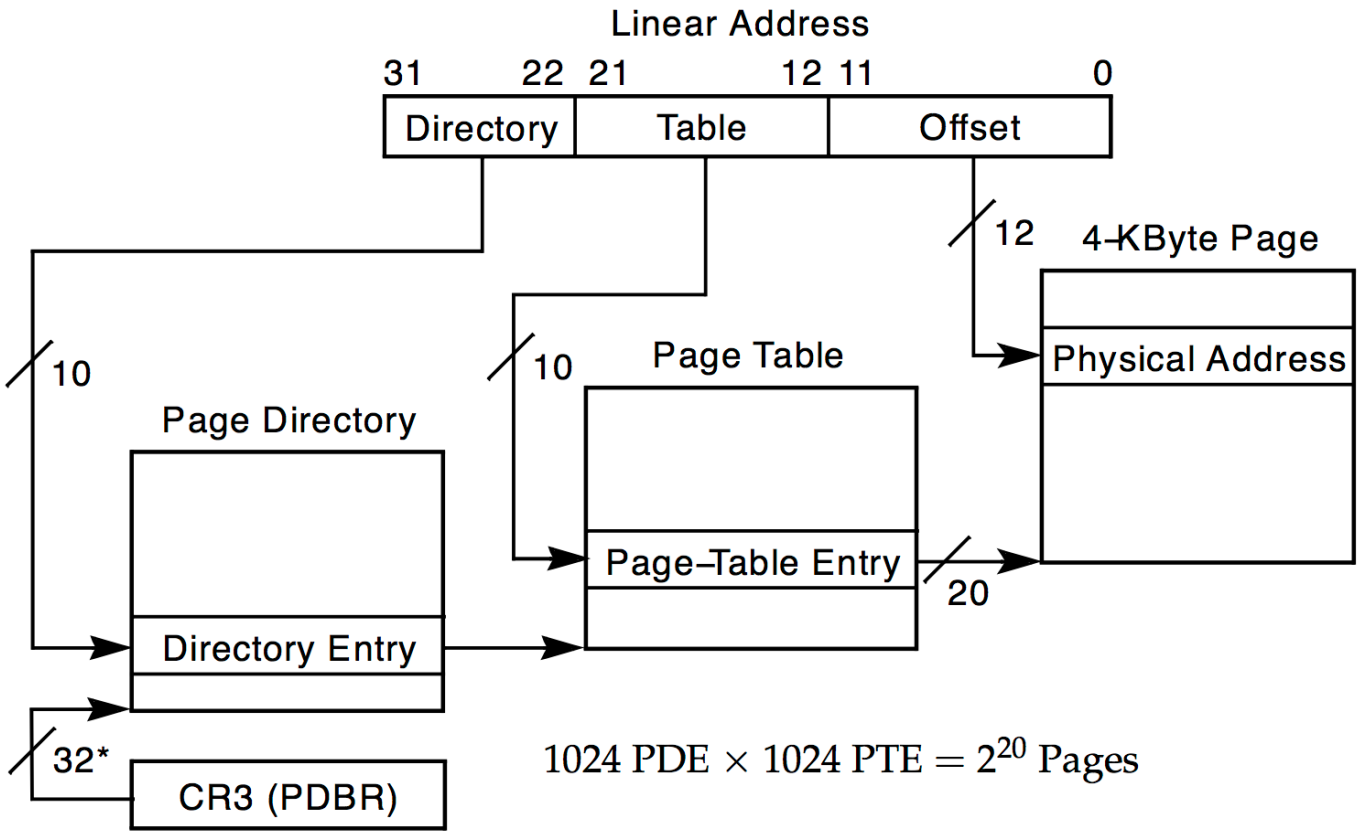
- **Comment implémenter la gestion de la mémoire virtuelle ?**
  - De façon purement logicielle ? Non : coût prohibitif en performances
  - De façon purement matérielle ? Non :
    - Complexité de la conception matérielle
    - Manque de flexibilité
  - Solution : compromis
    - Le matériel effectue automatiquement les traductions entre adresses virtuelles et adresses physiques
    - Le logiciel (noyau du système d'exploitation) configure les tables de traductions, gère les erreurs (accès invalides/interdits) et les défauts de page (chargement d'une page à partir du disque si nécessaire)
- **Memory management unit (MMU)**
  - Le module matériel qui gère la mémoire virtuelle – intégré au processeur

Support matériel pour la mémoire virtuelle (2)

- **Le noyau du système d'exploitation gère (alloue et configure) une table des pages par processus**
  - Ces tables sont stockées en mémoire centrale (RAM)
  - Un registre du processeur (dans le MMU) pointe vers la table des pages du processus courant
    - Ce registre est mis à jour lors d'un changement de contexte

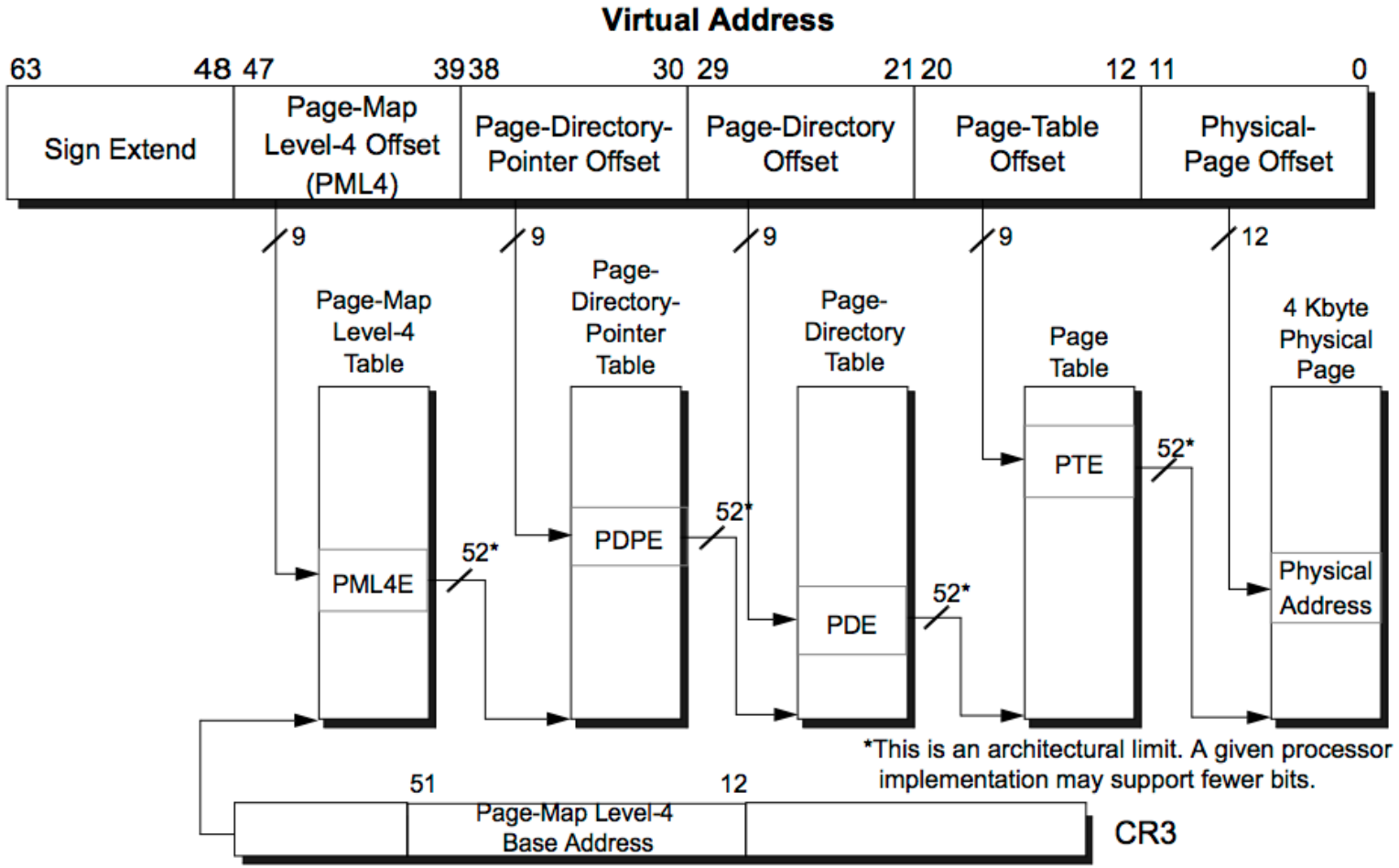
Support matériel pour la mémoire virtuelle (3)

- **Exemple : structure de la table des pages pour un processeur x86 32 bits**



Support matériel pour la mémoire virtuelle (4)

- **Exemple : structure de la table des pages pour un processeur x86 64 bits (seuls 48 à 52 bits sont actuellement utilisés)**



Support matériel pour la mémoire virtuelle (5)

- **Problème induit par la traduction d'une adresse (virtuelle => physique)**
  - Pour chaque instruction qui effectue un accès à la mémoire, on doit maintenant effectuer plusieurs accès à la mémoire supplémentaires (consultation de la table)
  - Impact négatif sensible sur les performances

- **Solution : TLB (*Translation lookaside buffer*)**
  - Un cache spécialisé qui stocke les couples d'adresses virtuelle/physique demandés récemment, dans l'espoir d'accès les demandes récurrentes
  - Intégré au MMU
  - Géré par le matériel (rempli à partir de la table des pages lorsque l'information n'est pas trouvée directement)
  - Doit être vidé lors d'un changement de contexte (pourquoi ?)
    - Sauf si les entrées contiennent un identifiant d'espace mémoire

Plan

- Hiérarchie mémoire
- Mémoire virtuelle
- Allocation dynamique de mémoire

Allocation dynamique de mémoire : introduction

L'allocation dynamique de mémoire est réalisée lors de l'exécution, par opposition à l'allocation statique (réservation d'une zone fixée, avant exécution).

Motivations

Besoins inconnus au moment de la compilation (structures de données dynamiques, procédures récursives)

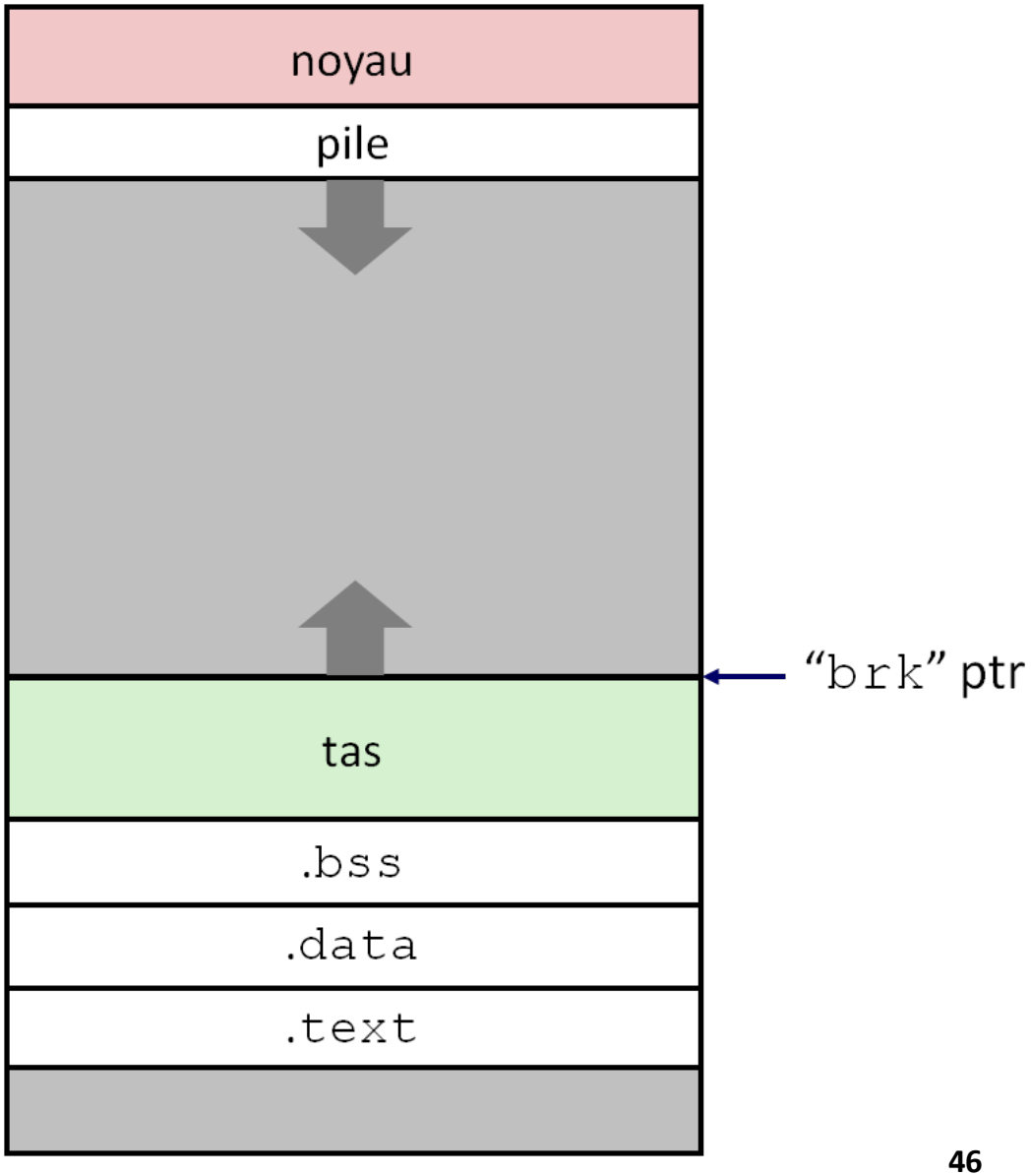
Modalités

Pour les structures dont la durée de vie obéit à une règle LIFO (*Last In, First Out*), où le dernier élément créé est le premier détruit, on utilise une pile (*stack*). Exemple : gestion des variables locales des procédures.

Dans tous les autres cas (pas d'informations sur la durée de vie), on utilise un tas (*heap*).
La pile et le tas occupent des segments différents dans la mémoire virtuelle d'un processus car leur mode de gestion est différent.

Allocation dynamique de mémoire : introduction

- **Allocateur mémoire**
  - Code exécuté en mode utilisateur (typiquement fourni par une bibliothèque)
  - Gère le contenu du tas
  - Si nécessaire, peut demander au noyau d'augmenter la taille du tas
    - Appel système **`sbrk()`**



Source : R. Bryant, D. O'Hallaron.
CSAPP 2nd edition

Allocation dynamique de mémoire : importance

L'allocation dynamique de mémoire est un aspect important

Incidence

L'allocation de mémoire peut avoir une forte influence sur les performances des applications, car :

- Beaucoup d'applications ont de forts besoins en mémoire (gestion de structures dynamiques)
- Une mauvaise gestion peut être très pénalisante à cause des différences importantes de temps d'accès en mémoire principale et secondaire (même si la mémoire virtuelle dissimule la distinction)

Problèmes

L'allocation dynamique de mémoire pose des problèmes délicats et elle est à l'origine d'erreurs difficiles à détecter (exemple : "fuite de mémoire", due à une mauvaise gestion des zone libres).

En conséquence, certains langages gèrent certains aspects de manière automatique/implicite, invisible aux applications (exemple : Java).

Allocation dynamique de mémoire : primitives en C

Rappel : Les primitives d'allocation de la mémoire (virtuelle) sous Unix sont *malloc()* et *free()*. Elles font partie de la bibliothèque C standard (*libc*).

***void *malloc(size_t size)* renvoie un pointeur vers un bloc de mémoire virtuelle de taille au moins égale à *size* (ajusté selon alignement ; en général frontière de double mot : 8 octets).**
Si erreur (par ex. pas assez de place disponible), *malloc()* renvoie NULL et affecte une valeur au code d'erreur *errno*.
Attention : la mémoire n'est pas initialisée. La primitive *calloc()* fonctionne comme *malloc()* mais initialise le bloc alloué à 0.

***void free(void *ptr)* doit être appelé avec une valeur de *ptr* rendue par un appel de *malloc()*. Son effet est de libérer le bloc alloué lors de cet appel.**
Attention : pour une autre valeur de *ptr*, l'effet de *free()* est indéterminé.

Allocation dynamique de mémoire : critères

Les qualités souhaitables pour une bonne gestion de la mémoire sont :

- **Performances (pour allocation et libération)**
  - **si possible, temps constant (indépendant de la taille, de l'ordre des requêtes, etc.)**
- **Bonne utilisation de l'espace disponible**
  - **réduire la fragmentation, réduire l'espace perdu**
- **Robustesse**
  - **vérifier que la libération porte bien sur une zone allouée**
  - **vérifier qu'on ne référence pas un espace non alloué**

Les réalisations de *malloc()* et *free()* ne satisfont pas toujours le dernier critère.

Allocation dynamique
Gestion explicite ou implicite des zones allouées

- **Gestion explicite :**
  - L'application alloue et doit libérer les zones
  - Exemple typique : C
- **Gestion implicite :**
  - L'application alloue les zones mais leur libération est automatique
  - Stratégie répandue dans les langages objet, fonctionnels ou de scripts (exemples : Java, ML, Lisp, Perl ...)
  - Stratégie adaptable à C/C++ mais avec des restrictions
  - Repose sur l'intervention d'un GC (*garbage collector* ou « ramasse-miettes »)
  - Il existe de nombreux types de GC
  - Simplifie et fiabilise le code applicatif mais dégrade généralement les performances

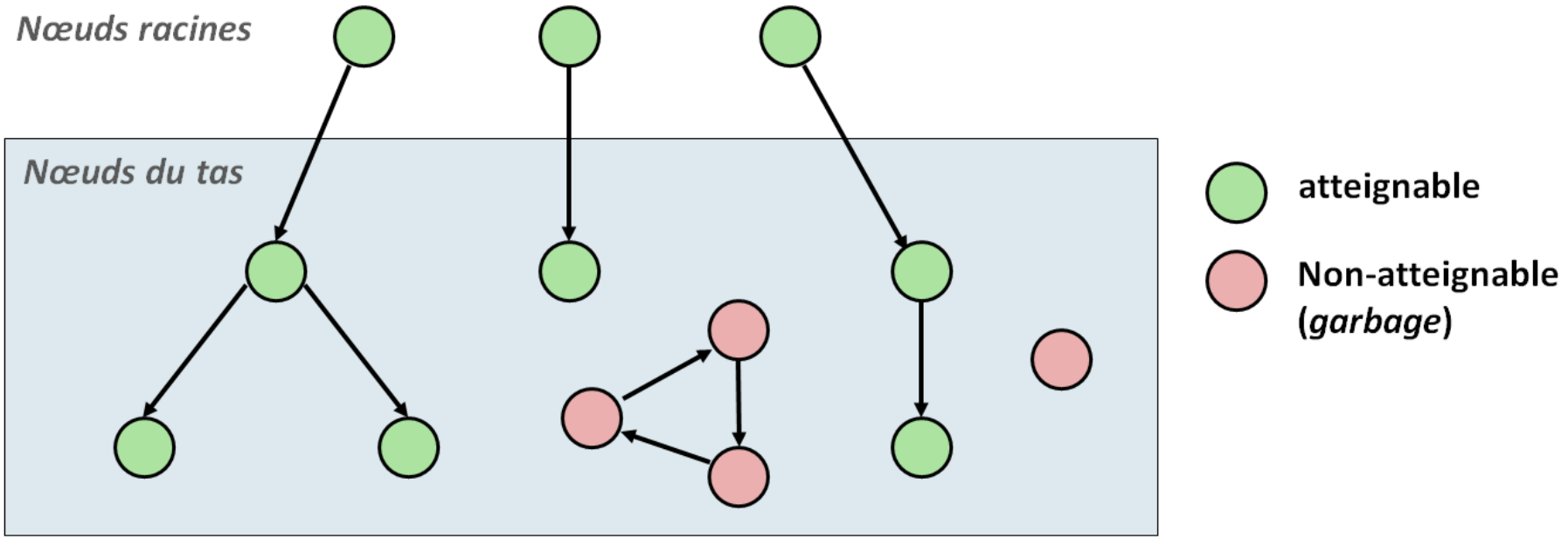
Principe d'un ramasse-miettes (1)

- **Idée : une zone de mémoire devient inutile si elle n'est plus accessible par l'application**

- **Hypothèses :**
  - On sait distinguer les objets de type « pointeurs » des autres objets
  - Un pointeur ne peut cibler que le début d'une zone mémoire
  - On ne peut pas « dissimuler » un pointeur (stockage d'une adresse dans un autre type de donnée)
  - NB : ces hypothèses ne sont pas vérifiées en C/C++

Principe d'un ramasse-miettes (2)

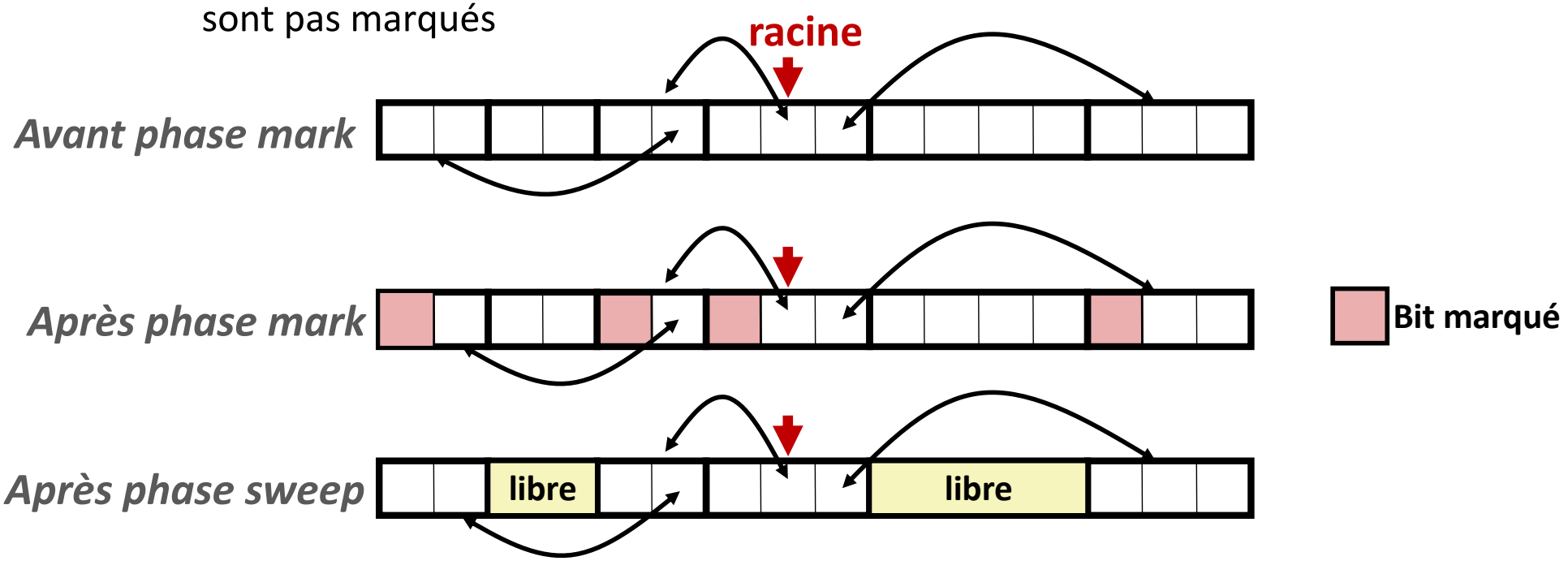
- **(On se concentre ici sur l'approche dite « traversante », l'une des plus utilisées)**
- **La mémoire est vue comme un graphe orienté**
  - Chaque zone allouée correspond à un nœud du graphe
  - Chaque pointeur correspond à un arc du graphe
  - Les emplacements mémoire situés hors du tas qui contiennent des pointeurs vers le tas sont appelés des nœuds racines (exemples : registres, variables globales, contenu de la pile)
  - Une zone est inutile (et donc récupérable) s'il n'existe aucun chemin permettant de l'atteindre à partir d'une racine



Source : R. Bryant, D. O'Hallaron. *CSAPP 2nd edition*

Un exemple (simplifié) d'algorithme pour ramasse-miette : *mark and sweep*

- **Utiliser *malloc* pour allouer tant qu'il y a de la place disponible**
- **Lorsqu'il n'y a plus de place libre, partir à la recherche des blocs de mémoire qui sont devenus inutilisés (inatteignables)**
- **Recherche :**
  - Utilise un bit spécial présent dans l'en-tête de chaque bloc
  - Phase *Mark* : parcours et marquage des blocs atteignables à partir des racines
  - Phase *Sweep* : scanner l'ensemble des blocs et récupérer (recycler) ceux qui ne sont pas marqués



Source : R. Bryant, D. O'Hallaron. *CSAPP 2nd edition*