

# **Gestion de la mémoire**

---

**Noel De Palma**  
Université Grenoble Alpes

**Ce cours est basé sur des transparents de Sacha Krakowiak/  
Renaud Lachaize**

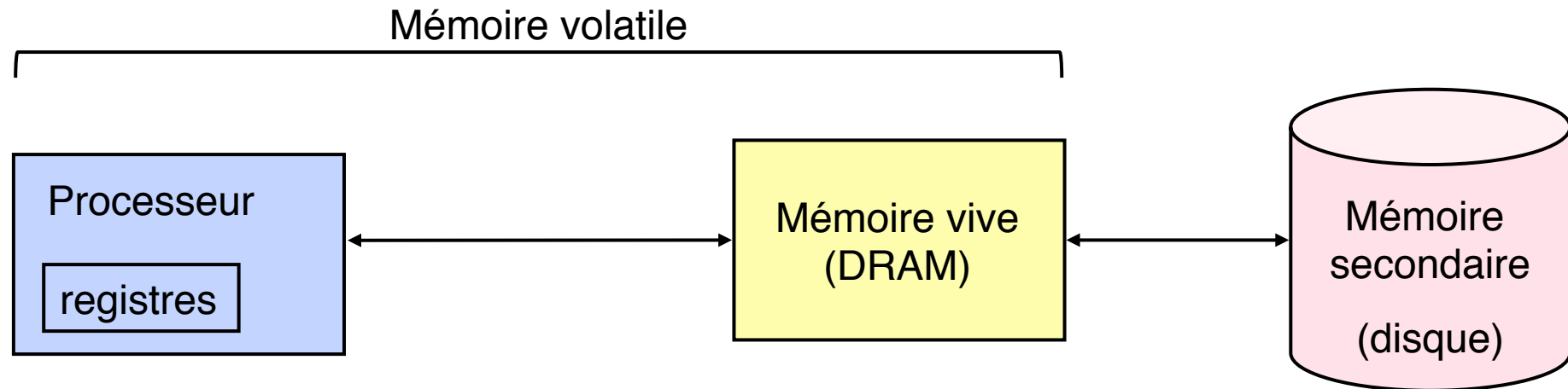
# Plan

---

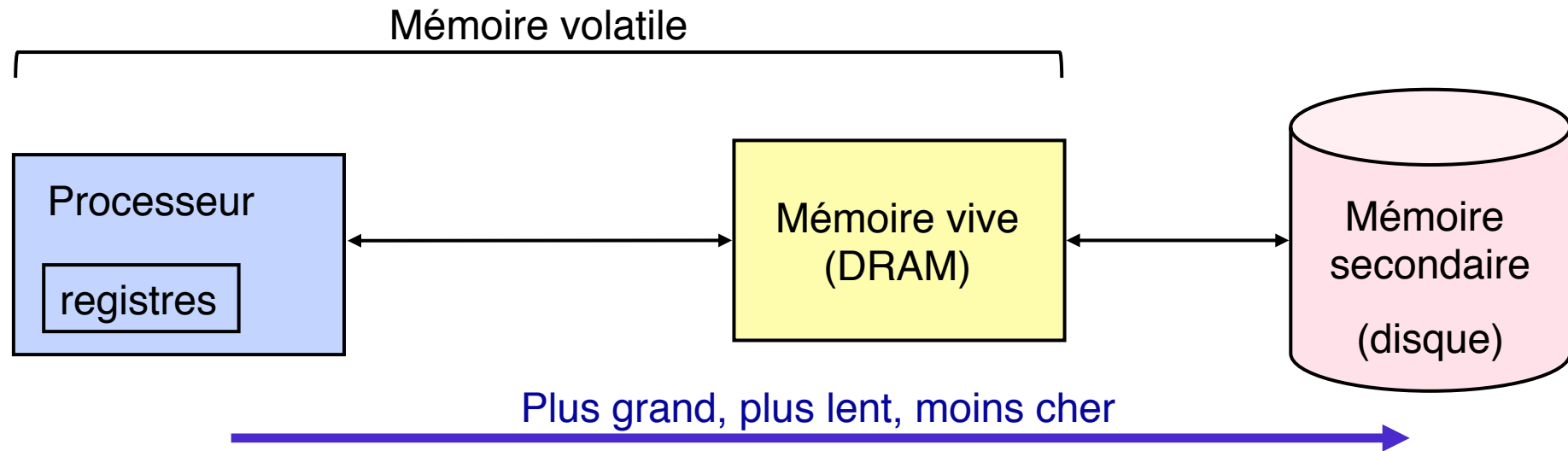
- **Hiérarchie de mémoire**
- **Mémoire virtuelle**
  - ◆ Motivation et principes
  - ◆ Réalisation
  - ◆ Couplage de fichiers
- **Allocation dynamique de mémoire**
- **Pièges de la gestion mémoire en langage C**

# Hiérarchie mémoire

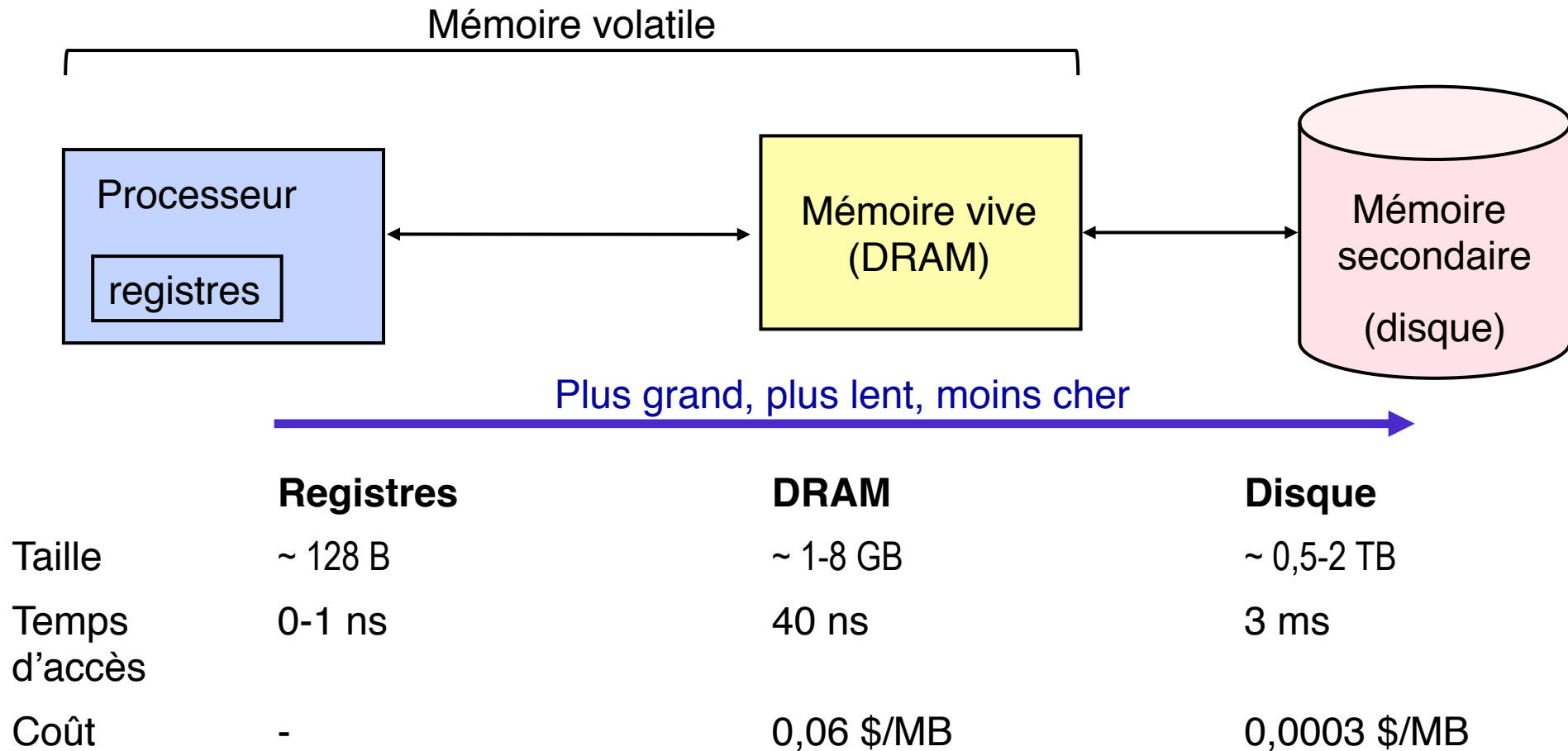
---



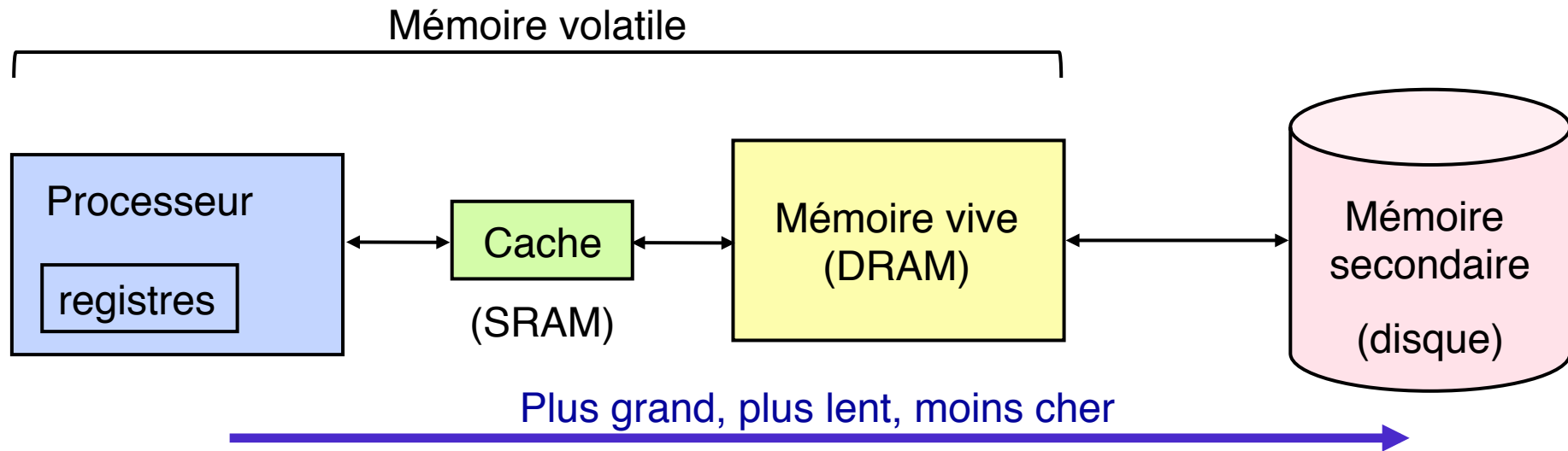
# Hiérarchie mémoire



# Hiérarchie mémoire

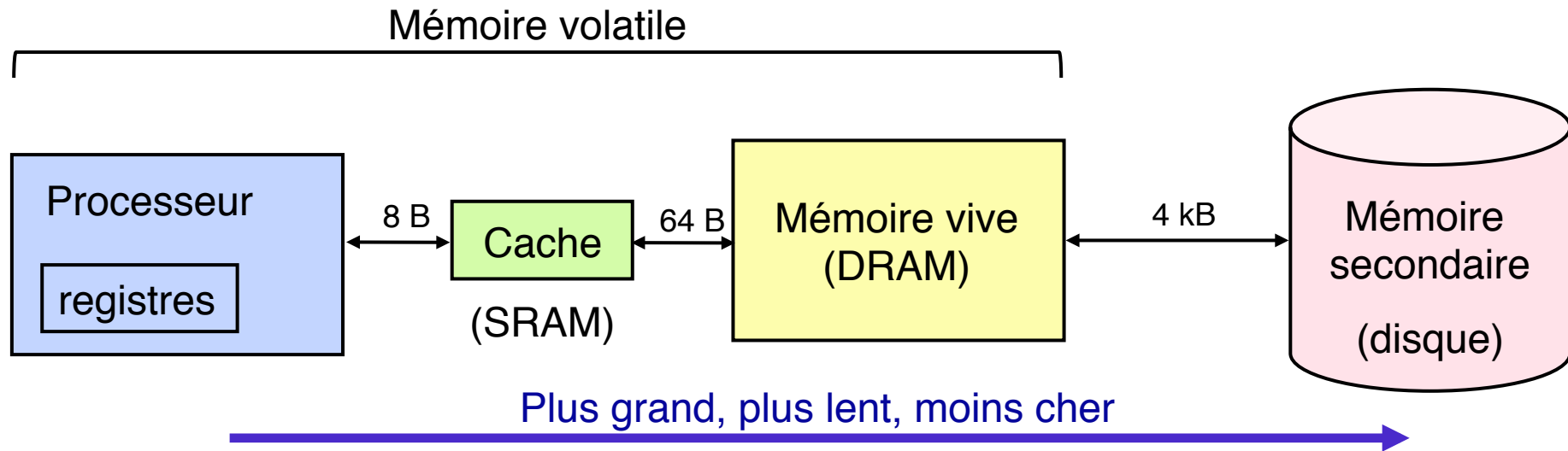


# Hiérarchie mémoire



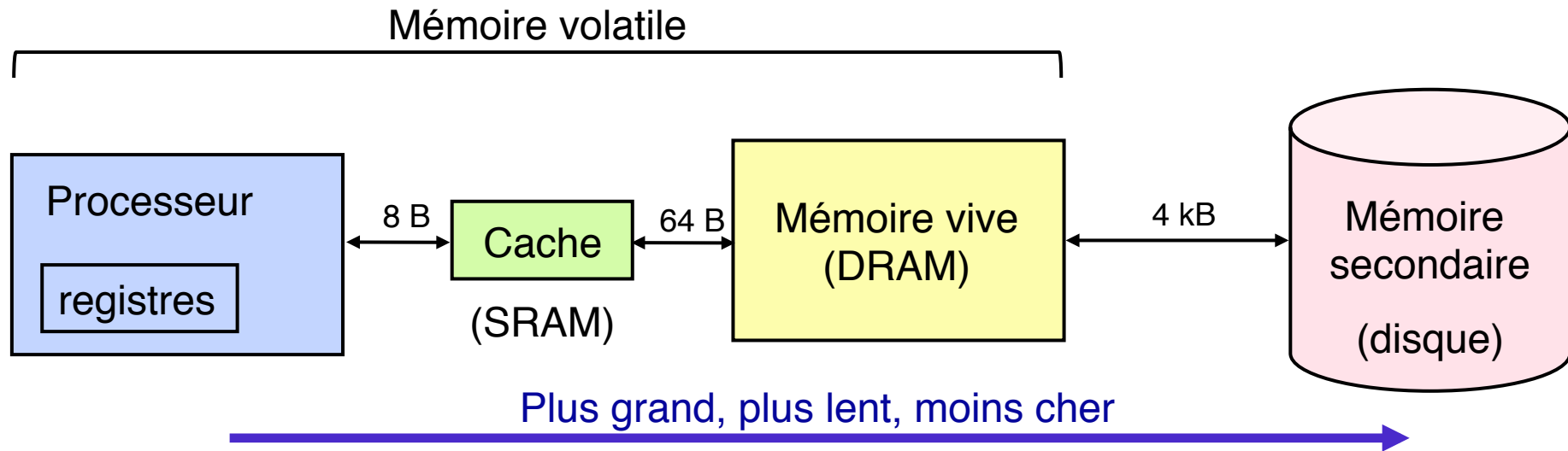
	<b>Registres</b>	<b>Cache</b>	<b>DRAM</b>	<b>Disque</b>
Taille	~ 128 B	~ 32-12 MB	~ 1-8 GB	~ 0,5-2 TB
Temps d'accès	0-1 ns	2-10 ns	40 ns	3 ms
Coût	-	60 \$/MB	0,06 \$/MB	0,0003 \$/MB

# 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 (2)

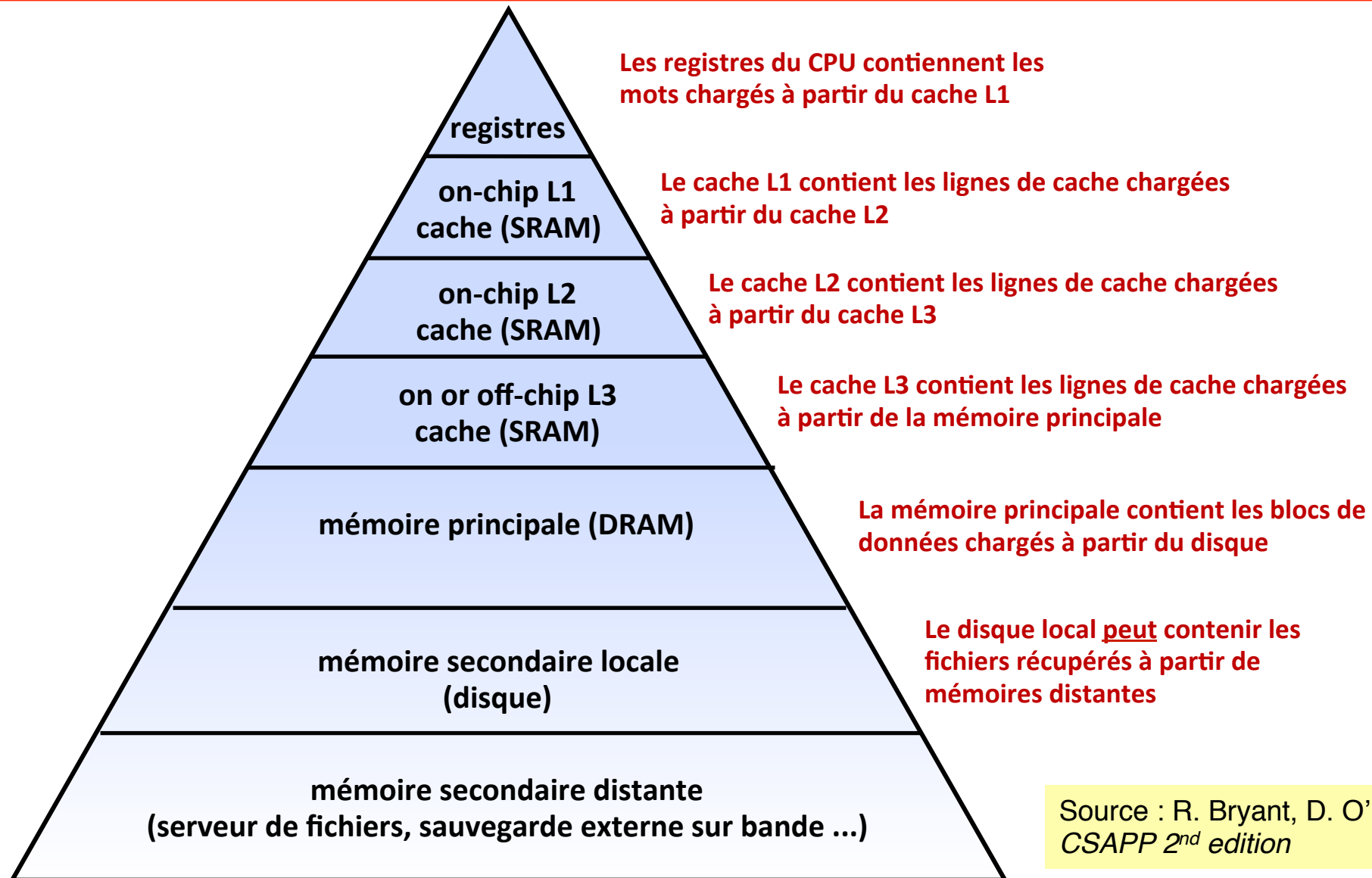


- L'essentiel des données est conservé en mémoire secondaire
- Les données sont chargées en mémoire principale et manipulées dans les registres du CPU
- La bonne gestion des transferts entre les différents niveaux de mémoire a un impact primordial sur les performances



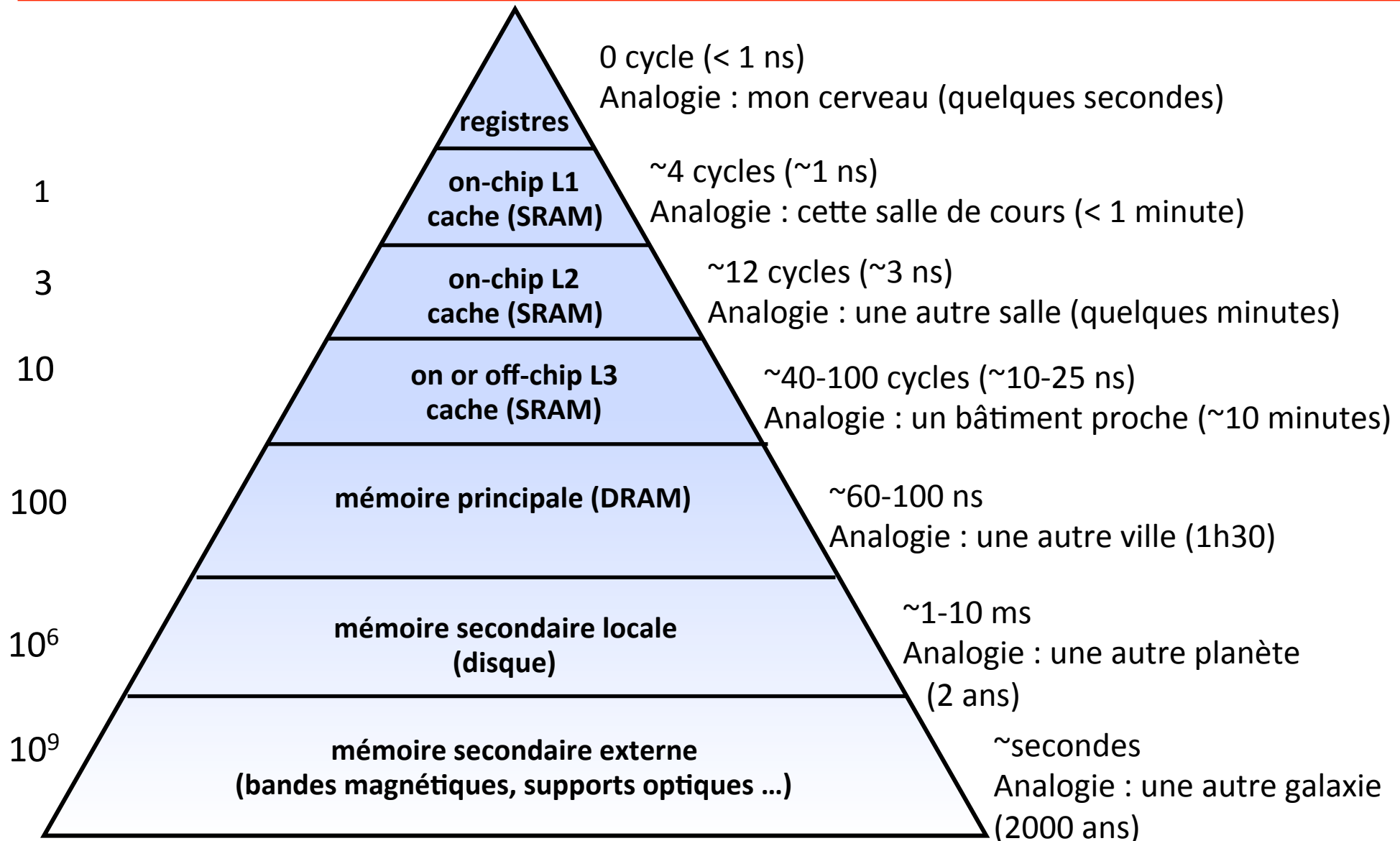
# Hiérarchie mémoire (3)

## Généralisation du principe



Source : R. Bryant, D. O'Hallaron.  
*CSAPP 2<sup>nd</sup> edition*

# Hiérarchie mémoire (5) Ordres de grandeur



# The memory hierarchy – An analogy

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 $\mu$ s	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	

# Notion de « cache »

---

## ■ Idée

- ◆ Améliorer les performances des accès aux données

## ■ 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 2<sup>nd</sup> edition*

# Plan

---

- **Hiérarchie de mémoire**
- **Mémoire virtuelle**
  - ◆ Motivation et principes
  - ◆ Réalisation
  - ◆ Couplage de fichiers
- **Allocation dynamique de mémoire**
- **Pièges de la gestion mémoire en langage C**

# 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

---

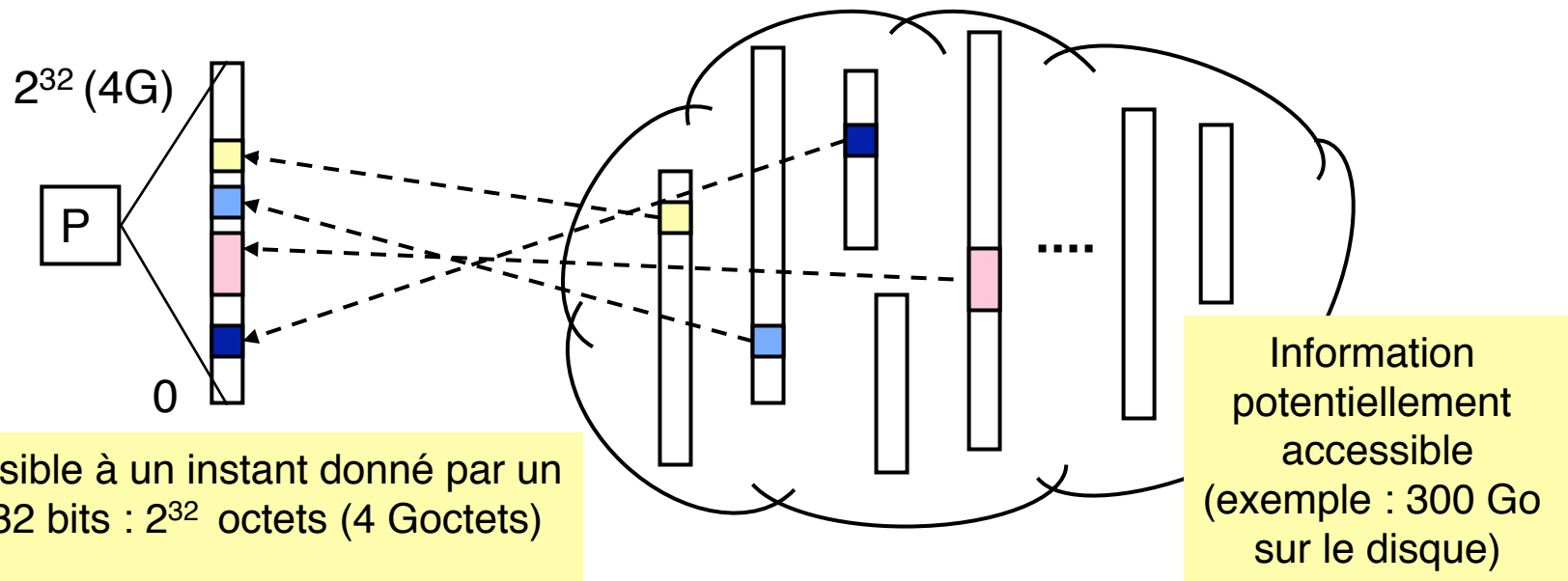
## ■ Pour lancer l'exécution d'un processus, on doit placer en mémoire les informations suivantes :

- ◆ 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 : Une “fenêtre d’adressage”

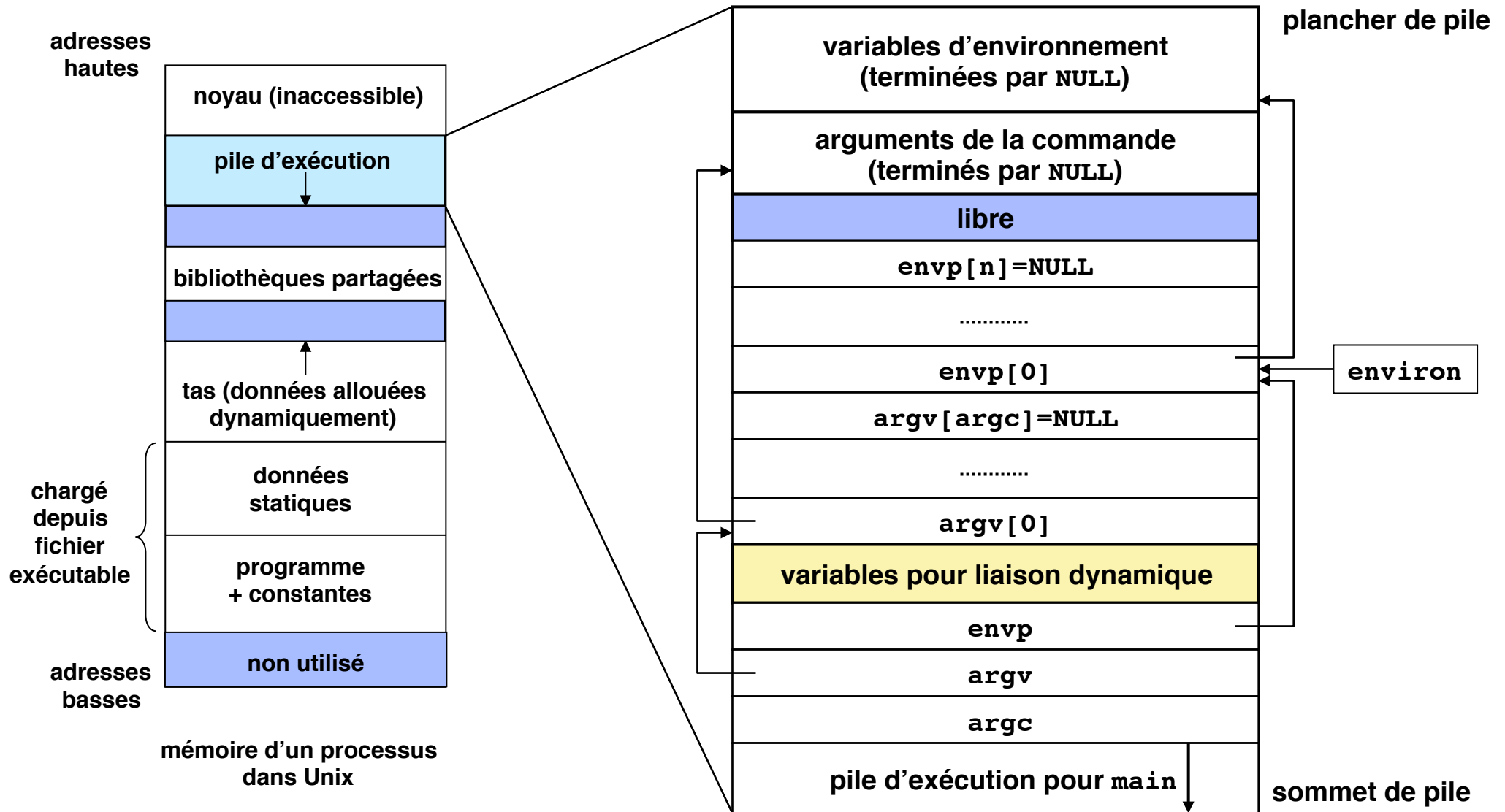
Un processus ne peut “voir”, à un instant donné, qu’un espace limité par la capacité d’adressage du processeur. Cet espace représente une fenêtre sur l’ensemble de l’information potentiellement accessible.



Deux aspects à considérer :

- **Logique** : déterminer et structurer, à tout moment, quelles sont les informations accessibles par un processus
- **Physique** : permettre l’accès effectif à ces informations lorsque cela est nécessaire

# 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 sont 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 bogue (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
    - ❖ 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 est associé 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 :
  - ❖ Éviter des problèmes de placement (adresses identiques, manque de contiguïté)
  - ❖ Contrôler l'accès aux informations
  - ❖ Éviter le stockage d'informations redondantes



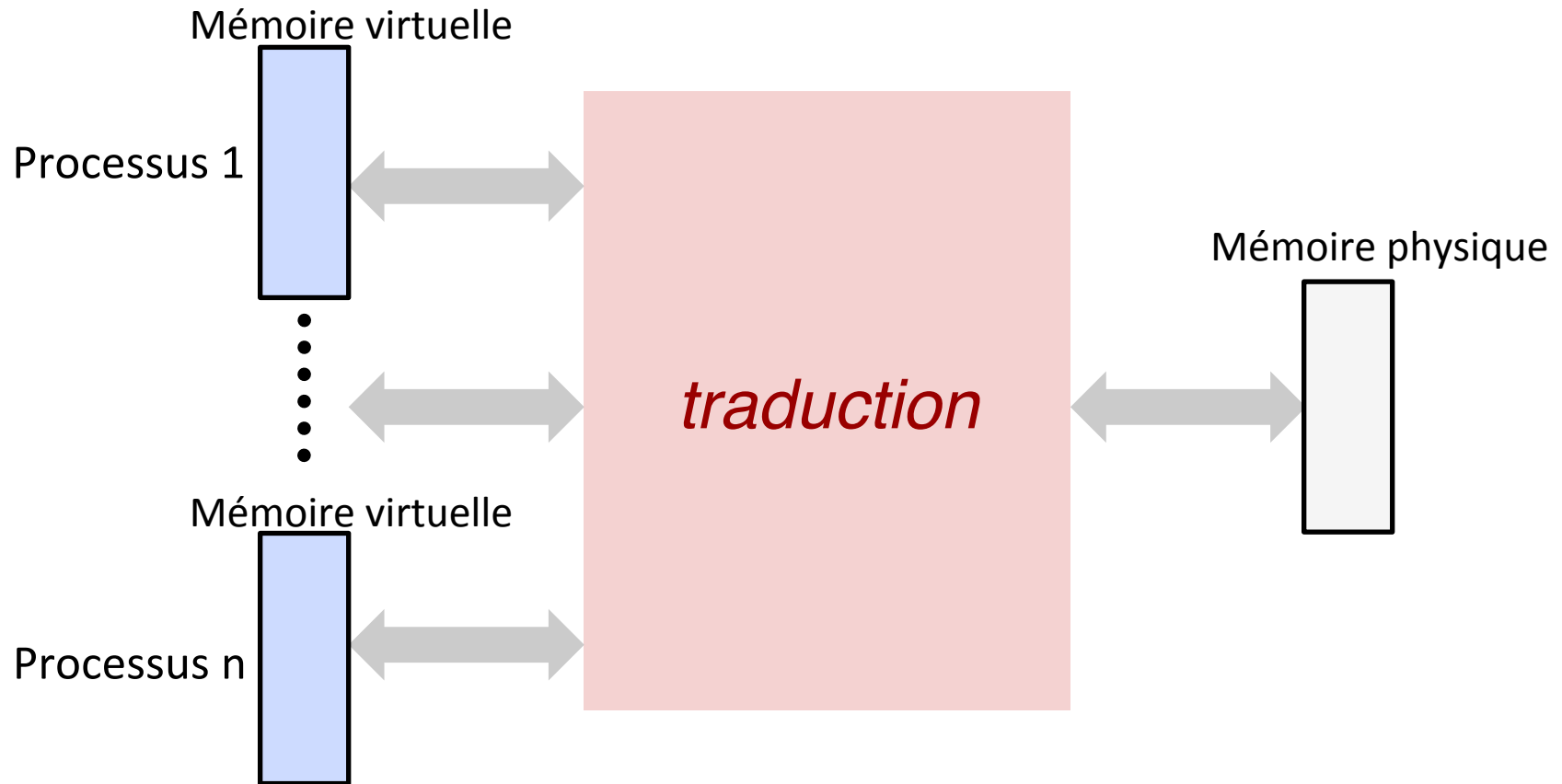
# Mémoire virtuelle : Principe de la solution (2)

---

## ■ 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 : Principe de la solution (3)



Source : R. Bryant,  
D. O'Hallaron.  
*CSAPP 2<sup>nd</sup> edition*

# Mémoire virtuelle : Principe 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

# Plan

---

- **Hiérarchie de mémoire**
- **Mémoire virtuelle**
  - ◆ Motivation et principes
  - ◆ Réalisation
  - ◆ Couplage de fichiers
- **Allocation dynamique de mémoire**
- **Pièges de la gestion mémoire en langage C**

# Principe de la mémoire virtuelle paginée (1)

---

La mémoire principale et le disque sont organisés comme un ensemble de **pages** de **taille fixe** (de l'ordre de 4 à 8 Koctets).

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)

À un instant donné, la mémoire physique ne peut contenir qu'un petit sous-ensemble des pages constituant l'ensemble des mémoires virtuelles. Mais ces pages sont toutes présentes sur le disque.


Partage des rôles entre le système d'exploitation et les utilisateurs

- Les utilisateurs **définissent le contenu** des mémoires virtuelles
- Le système d'exploitation garantit que toute page virtuelle (programme ou données) utilisée par le processeur **sera amenée en temps utile** du disque en mémoire physique. Ces transferts sont invisibles aux utilisateurs.

# 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 ?

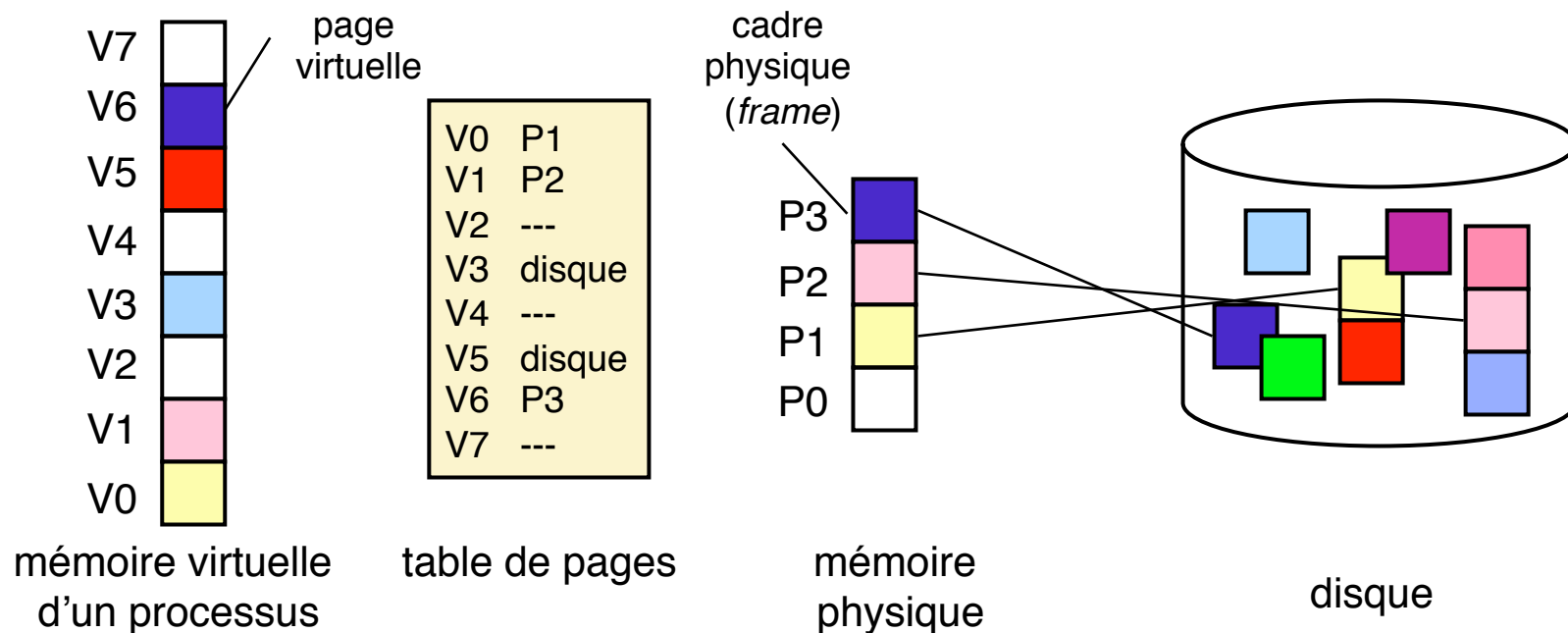
- ◆ Commodité de gestion (les cadres sont interchangeables)
- ◆ Efficacité (taille des structures de données, amortissement du coût)

## ■ 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)

Principe général seulement, sans entrer dans les détails, qui seront vus en M1



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

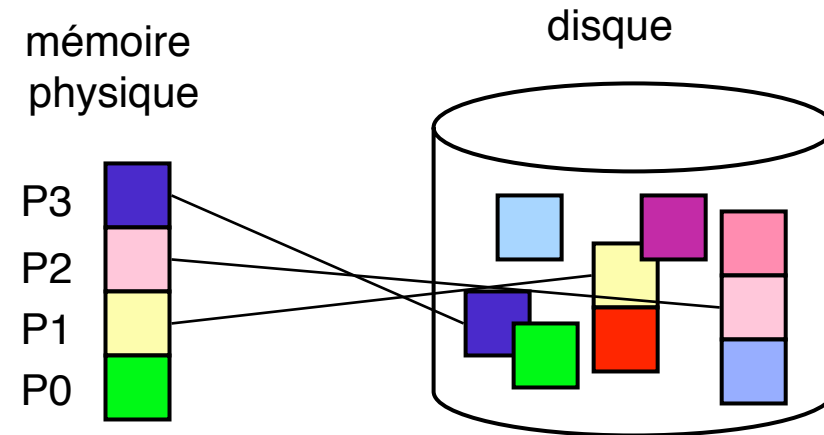
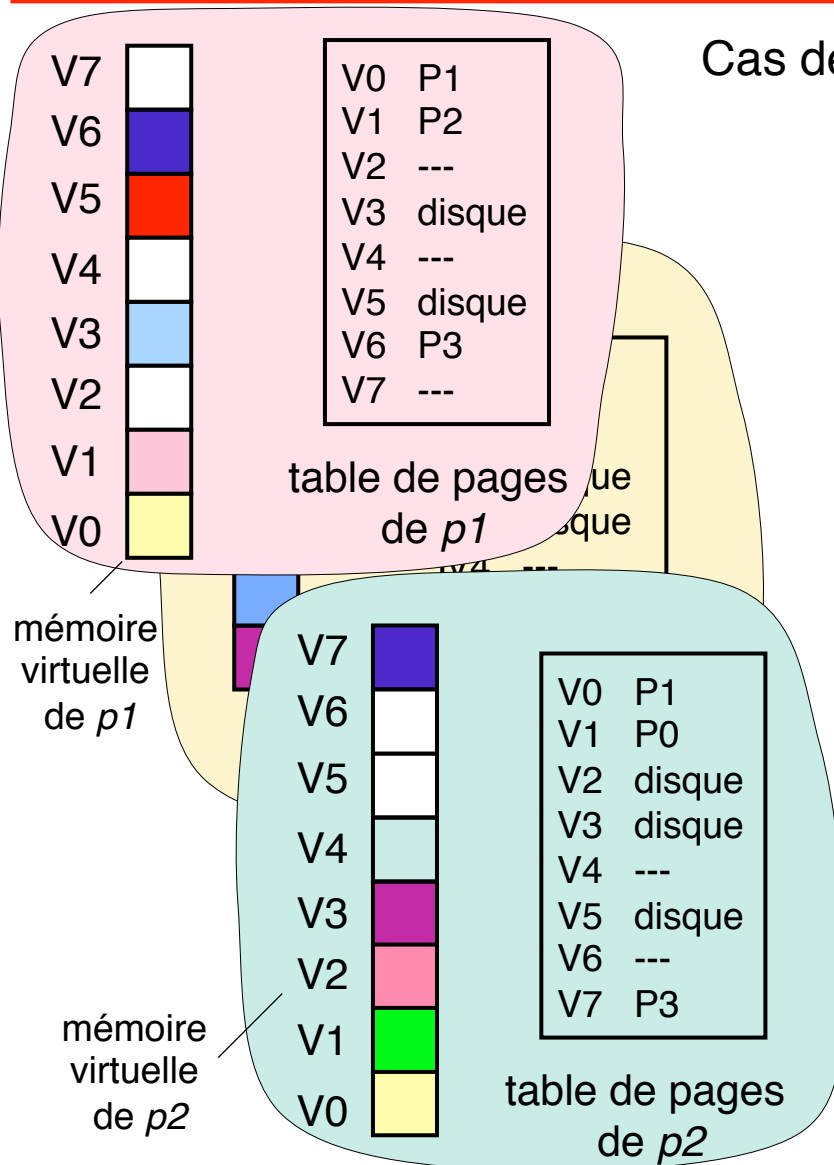
Une page virtuelle dont le contenu est en mémoire physique est immédiatement accessible.

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

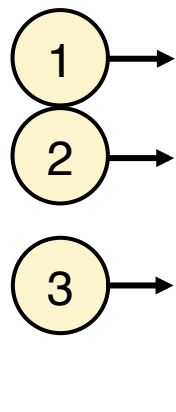


Chaque processus possède sa propre mémoire virtuelle, indépendante de celle des autres processus (il a donc sa propre table de pages). Des processus peuvent partager des pages, à la même adresse (page V0) ou à des adresses différentes (page V6 de  $p1$  et V7 de  $p2$ )



# Pagination à la demande (très simplifié)

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)



V0	P1
V1	P2
V2	---
V3	disque
V4	---
V5	disque
V6	P3
V7	---

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

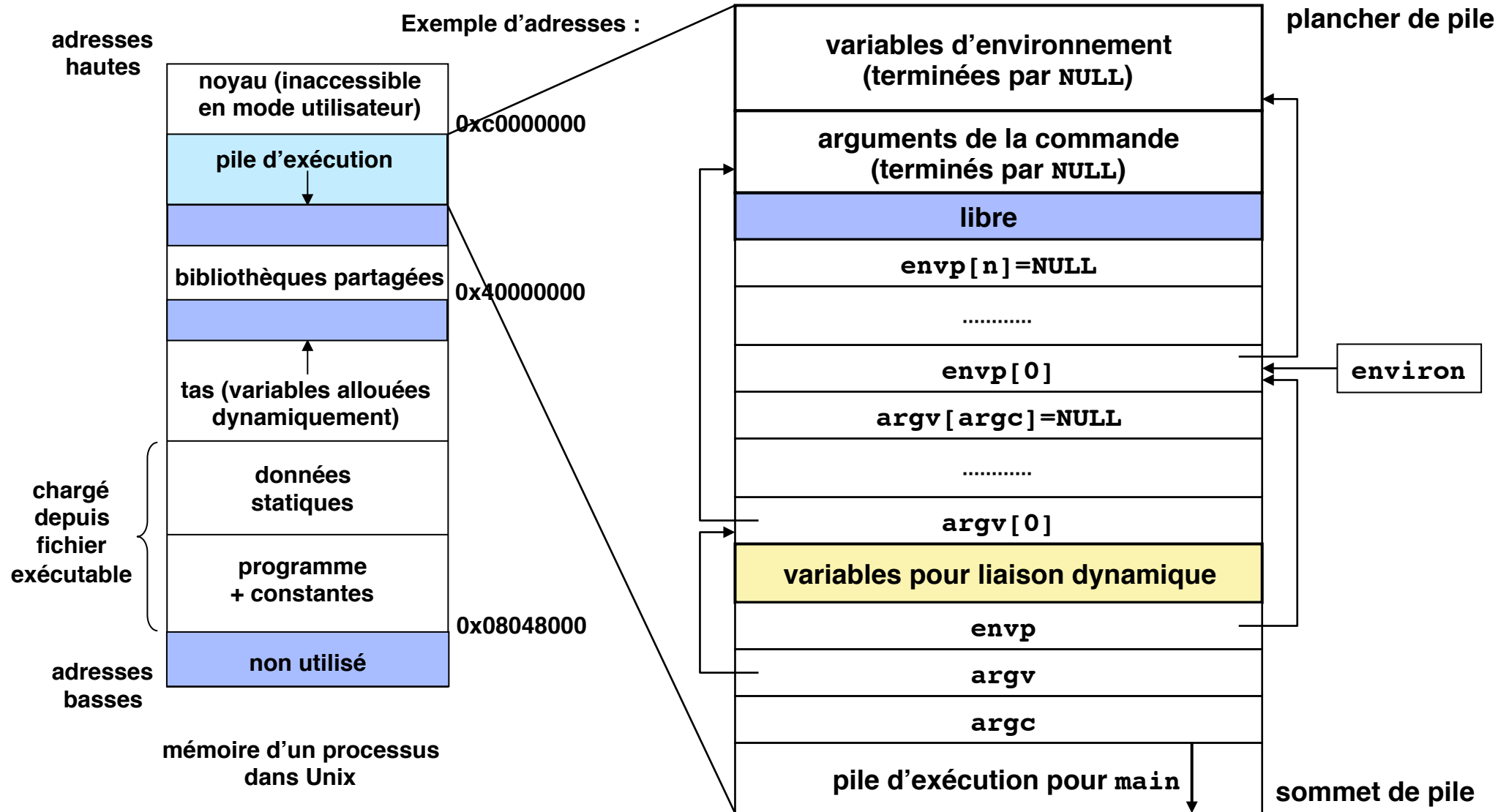
**Cas 3** (exemple : V5) : la page correspondante est sur disque (à une adresse indiquée). Il y a **défaut de page**.

## Traitement du défaut de page

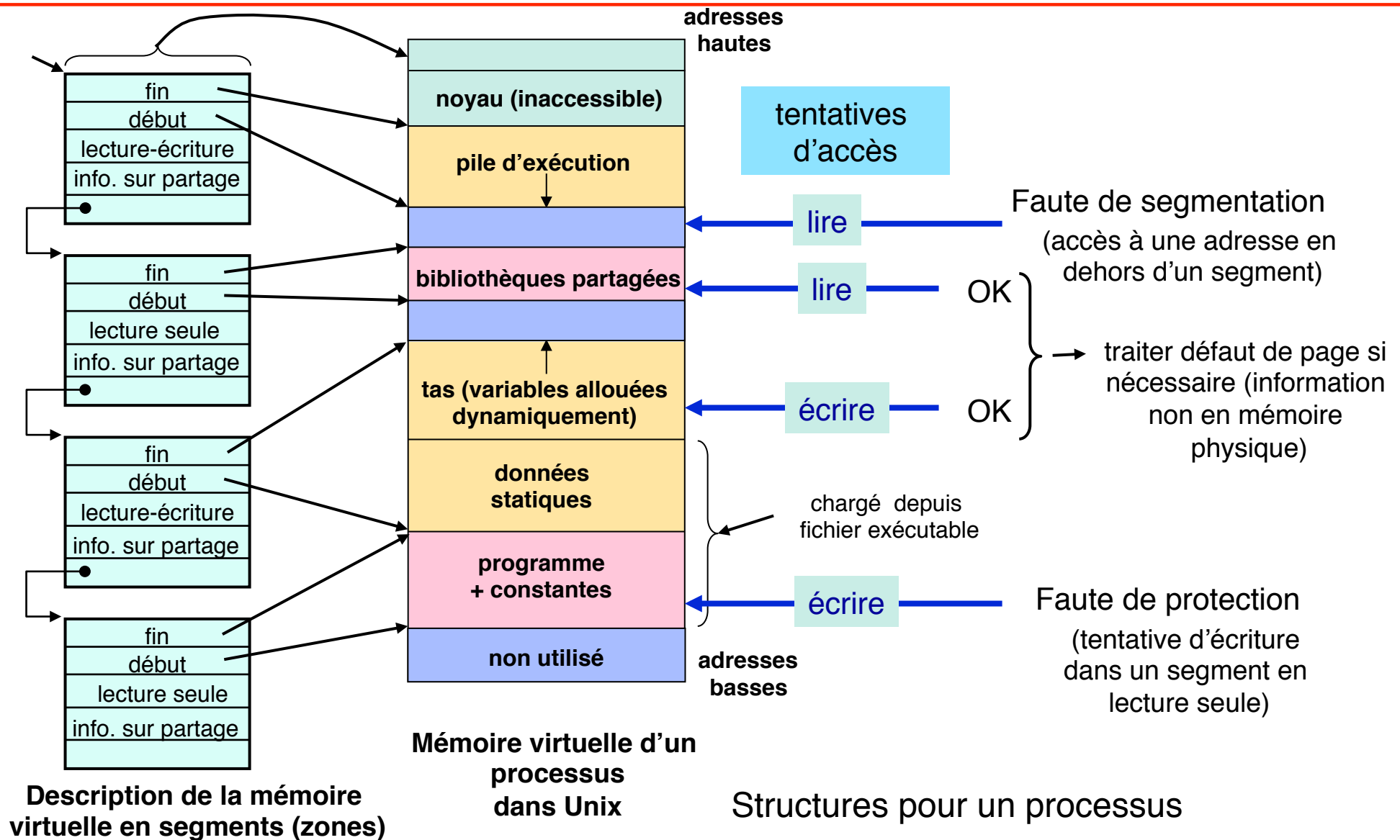
Le processus demandeur est bloqué (le processeur est donc alloué à un autre processus). Le système d'exploitation trouve un cadre libre en mémoire physique et y charge la page depuis le disque. Quand la page est chargée, la table des pages est mise à jour et le processus est réveillé.

Les défauts de page sont très coûteux. Il faut donc réduire leur nombre (voir cours de M1)

# Organisation et protection de la mémoire virtuelle (1)

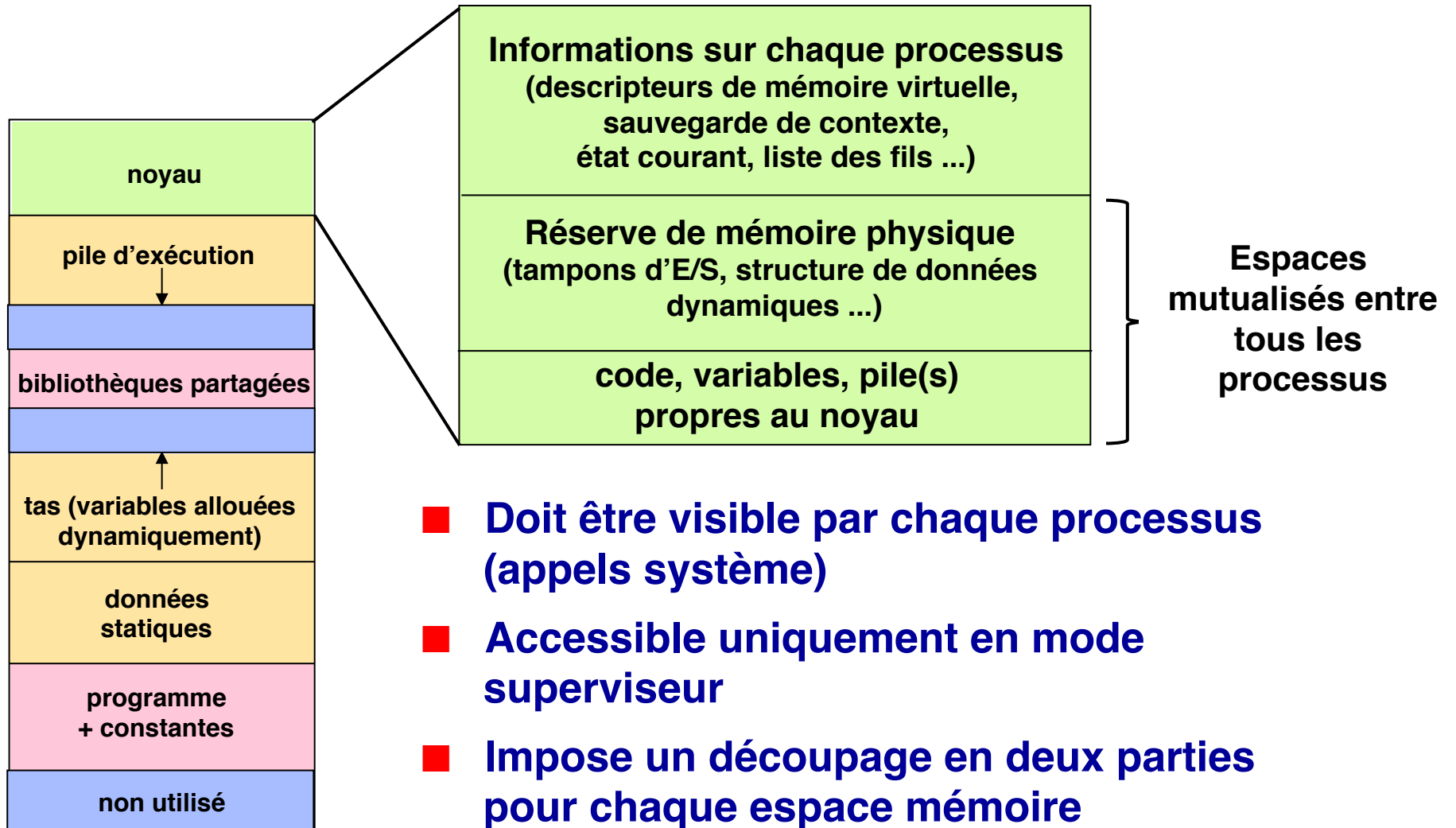


# Organisation et protection de la mémoire virtuelle (2)



# Organisation et protection de la mémoire virtuelle (3)

## Le noyau du système d'exploitation



# Plan

---

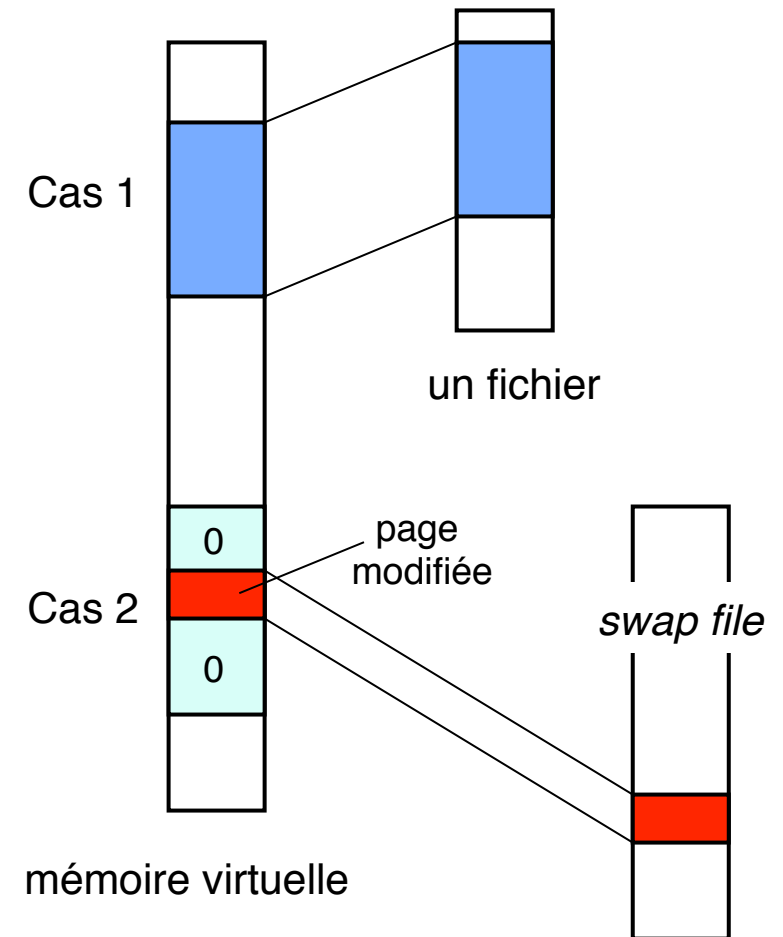
- **Hiérarchie de mémoire**
- **Mémoire virtuelle**
  - ◆ Motivation et principes
  - ◆ Réalisation
  - ◆ Couplage de fichiers
- **Allocation dynamique de mémoire**
- **Pièges de la gestion mémoire en langage C**

# 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 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)
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. Si elle est modifiée, son contenu est copié sur un fichier commun de réserve (*swap file*). Un tel segment est dit "0-demande" puisqu'il n'y a pas initialement de fichier qui lui soit associé.

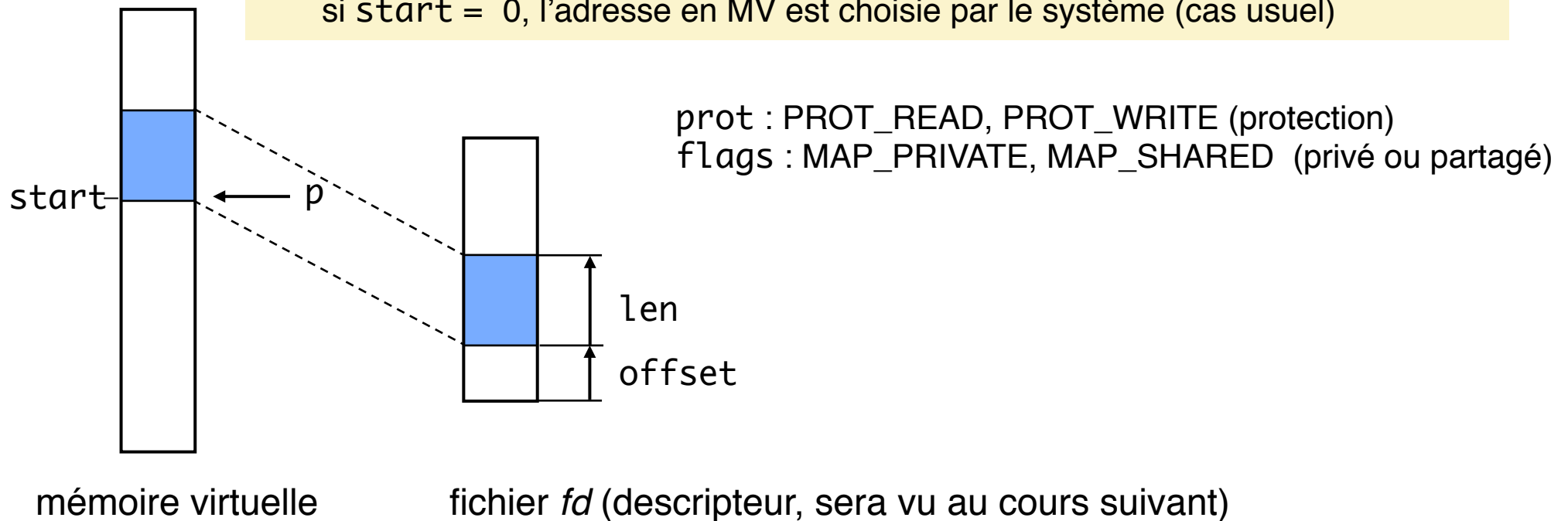


# Couplage entre mémoire virtuelle et fichiers

L'opération `mmap()` permet d'associer une zone de mémoire virtuelle à une zone de fichier. L'intérêt est de fournir un accès direct au contenu du fichier (via la mémoire virtuelle), **sans réaliser d'entrée-sortie explicite dans le code d'un programme** (voir cours suivant). Une page ne sera amenée en mémoire que si elle est référencée.

```
Usage : p = void *mmap(void *start, int len, int prot,
                      int flags, int fd, int offset)
```

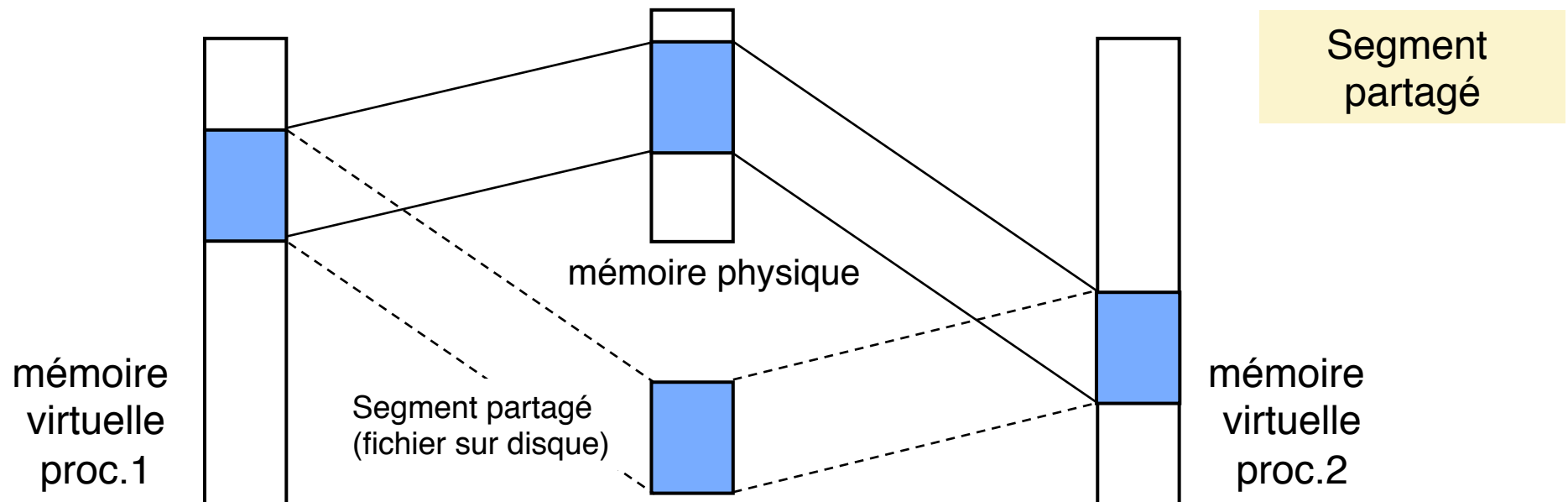
si `start = 0`, l'adresse en MV est choisie par le système (cas usuel)



# Partage de segments entre mémoires virtuelles (1)

Un segment (en pratique une zone de fichier, voir *mmap*) peut être associé à **plusieurs** mémoires virtuelles. Pour régler le problème du partage, on définit deux modes possibles d'association des segments en mémoire virtuelle : **partagé** ou **privé**.

Un segment partagé existe en **un seul** exemplaire en mémoire physique. Il peut être associé à une zone de la mémoire virtuelle de plusieurs processus. Toute modification par un processus devient visible à tous les autres (et est reportée sur disque).



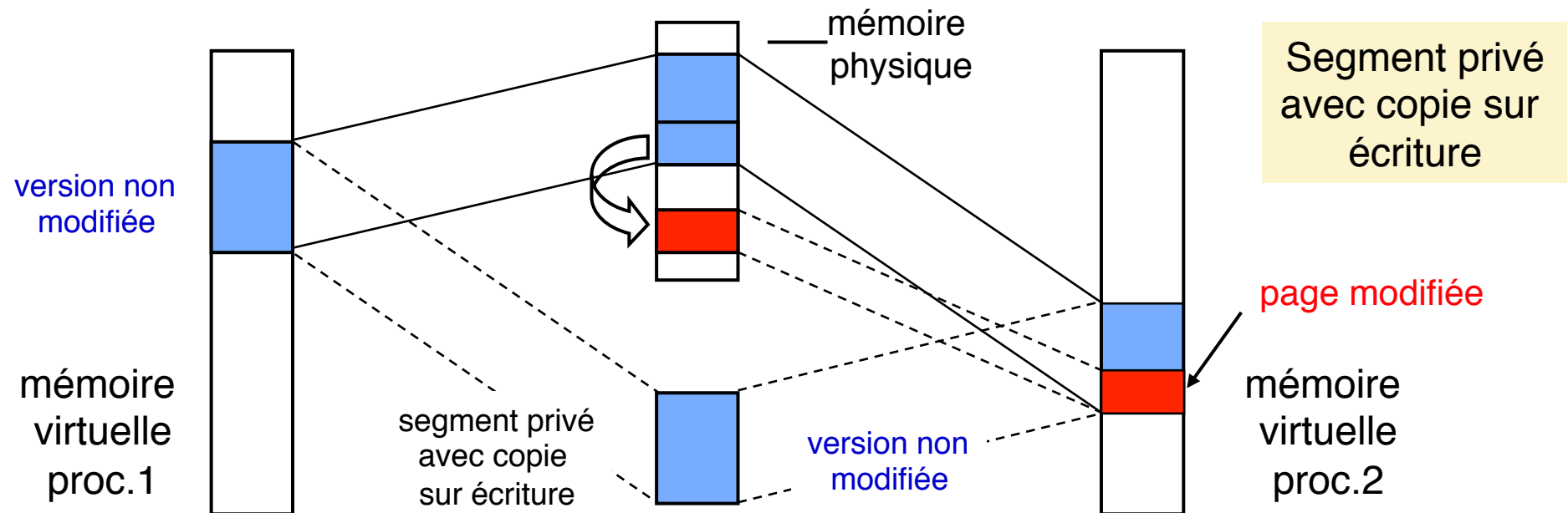


## Partage de segments entre mémoires virtuelles (2)

Un **segment privé** se comporte comme un segment partagé tant qu'il n'est modifié par aucun **processus** : il existe initialement en un seul exemplaire en mémoire physique et peut être associé à une zone de la mémoire virtuelle de plusieurs processus.

Si une page du segment est modifiée par un processus, une nouvelle page est allouée en mémoire physique et la modification est reportée sur cette seule page. La modification **n'est pas** visible aux autres processus, qui continuent à voir la version initiale. Même chose pour toute nouvelle modification par l'un des processus. Cette technique est la **copie sur écriture** (*copy on write*).

Cette technique est notamment utilisée pour réaliser *fork()*.



## Comment fonctionne *fork()* ?

---

Rappel : *fork()* crée un processus dont la mémoire virtuelle est initialement une copie conforme de celle de son père. En fait, **il n'y a pas recopie physique du contenu de la mémoire**, mais seulement recopie des descripteurs de segments et de la table des pages.

Le problème est maintenant de permettre aux deux mémoires d'évoluer indépendamment, tout en permettant le partage de parties communes non modifiées.

Pour cela, chaque segment des deux mémoires virtuelles est associé dans le mode **privé avec copie sur écriture**. Les deux mémoires peuvent à présent évoluer indépendamment à partir d'un état initial commun, en minimisant le nombre de pages utilisées.

Dans la pratique, chez le fils, *fork()* est généralement suivi d'*exec()*, qui modifie la mémoire virtuelle du fils en y associant un nouveau fichier exécutable.

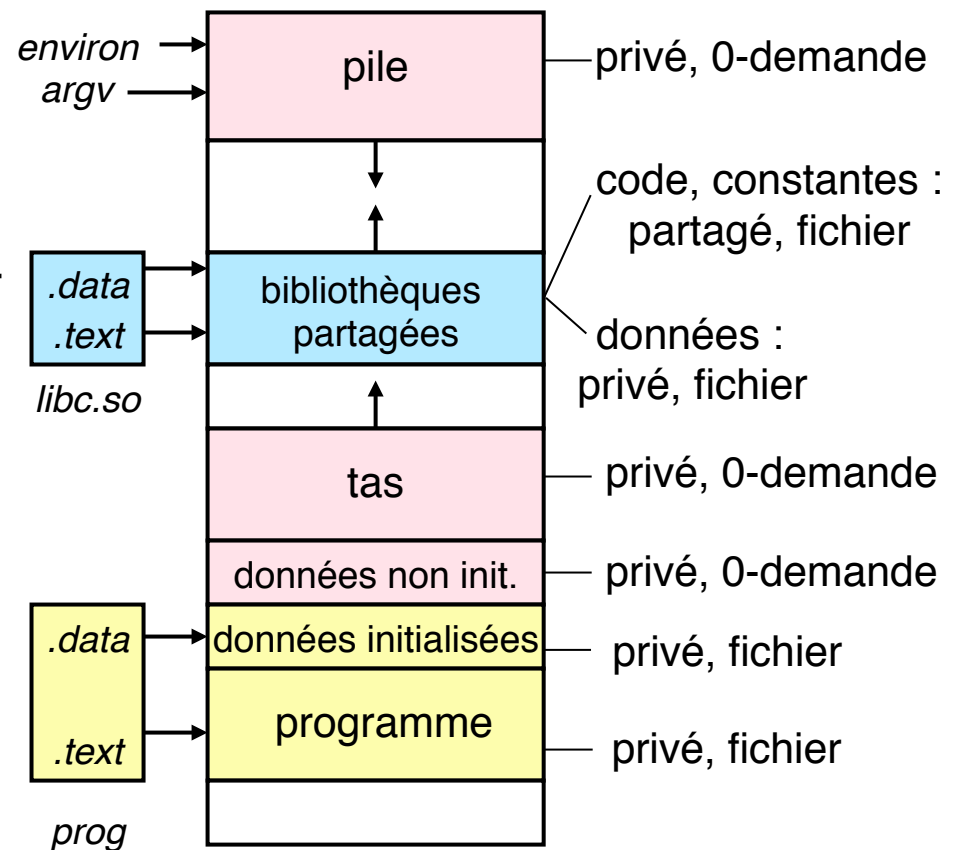
# Comment fonctionne `exec()` ?

Exemple : `execve("prog", argv, environ)` ;

Place en mémoire virtuelle le programme contenu dans le fichier *prog*, avec la zone de paramètres *argv* et la zone d'environnement *environ*. Puis lance l'exécution de *prog*.

## Fonctionnement interne :

- Supprime les descriptions de zones existantes
- Crée des descriptions de zones pour les zones *.text* (le programme exécutable), *.data* et *.bss* (les données initialisées ou non) et pour la pile. Ces zones pile et données sont privées avec copie sur écriture.
- Si nécessaire, associe les bibliothèques dynamiques (comme *libc.so*) dans la zone correspondante de mémoire virtuelle.
- Initialise le compteur ordinal (au point d'entrée du programme exécutable *prog*, en pratique pointe vers le *main* dans un programme C).



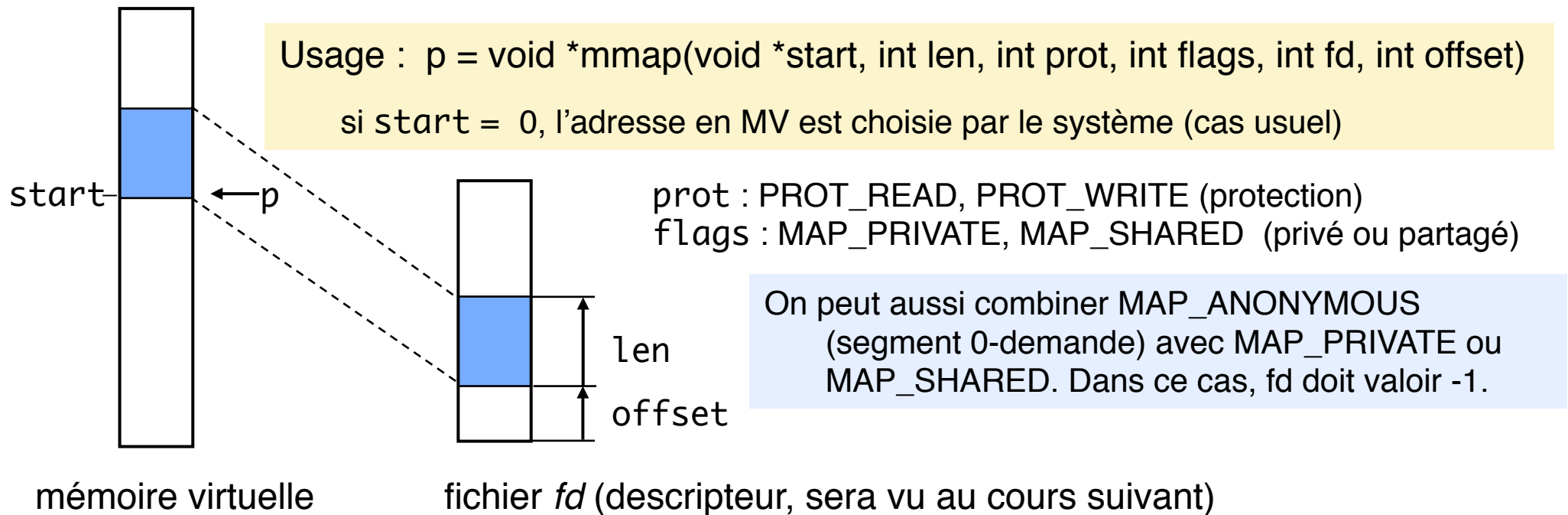
# Couplage entre mémoire virtuelle et fichiers

Le système utilise le mécanisme de couplage mémoire pour ses besoins internes (notamment lors de la création d'un processus – voir plus loin).

Ce mécanisme est également mis à disposition des applications via la primitive *mmap()*.

La primitive *mmap()* permet d'associer une zone de mémoire virtuelle à une zone de fichier. L'intérêt est de fournir un accès direct au contenu du fichier (via la mémoire virtuelle), **sans réaliser d'entrée-sortie explicite dans le code d'un programme** (voir cours suivant).

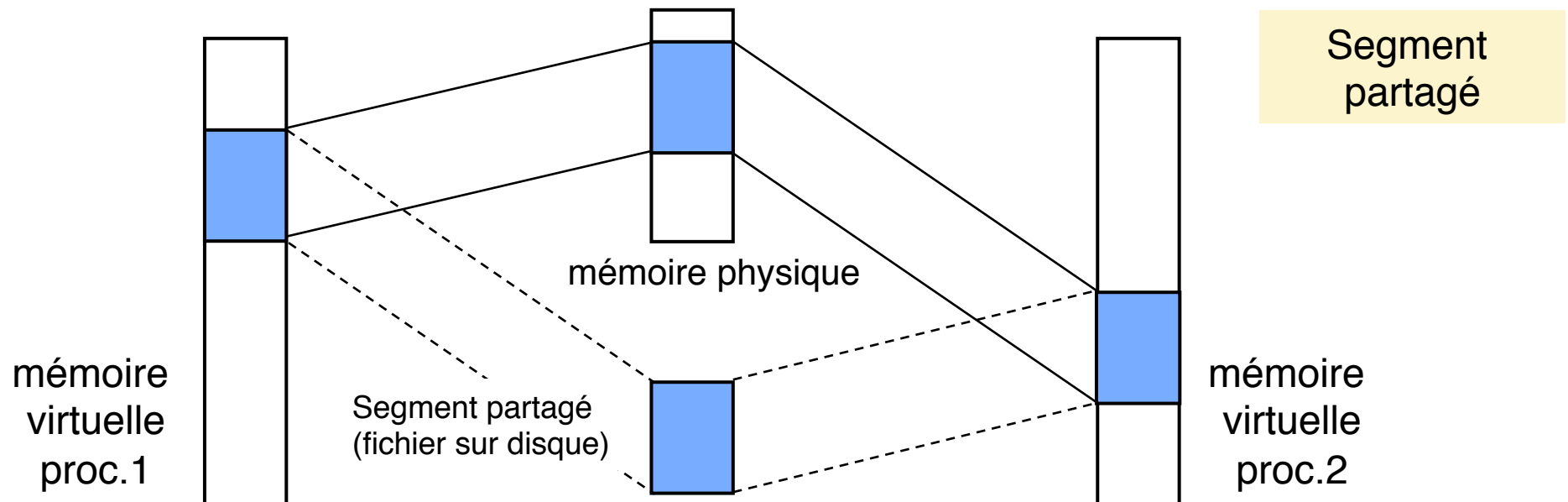
Une page ne sera amenée en mémoire que si elle est référencée.



# Partage de segments entre mémoires virtuelles (1)

Un segment (en pratique une zone de fichier, voir *mmap*) peut être associé à **plusieurs** mémoires virtuelles. Pour régler le problème du partage, on définit deux modes possibles d'association des segments en mémoire virtuelle : **partagé** ou **privé**.

Un segment partagé existe en **un seul** exemplaire en mémoire physique. Il peut être associé à une zone de la mémoire virtuelle de plusieurs processus. Toute modification par un processus devient visible à tous les autres (et est reportée sur disque).

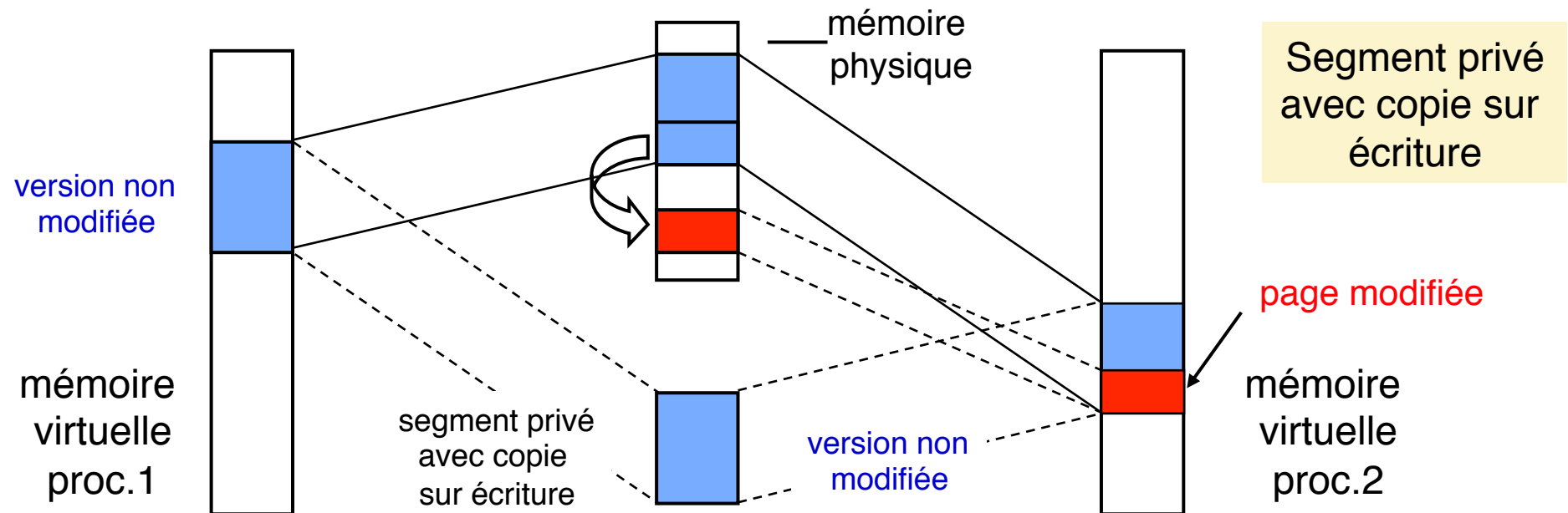


## Partage de segments entre mémoires virtuelles (2)

Un **segment privé** se comporte comme un segment partagé tant qu'il n'est modifié par aucun **processus** : il existe initialement en un seul exemplaire en mémoire physique et peut être associé à une zone de la mémoire virtuelle de plusieurs processus.

Si une page du segment est modifiée par un processus, une nouvelle page est allouée en mémoire physique et la modification est reportée sur cette seule page. La modification **n'est pas** visible aux autres processus, qui continuent à voir la version initiale. Même chose pour toute nouvelle modification par l'un des processus. Cette technique est la **copie sur écriture** (*copy on write*).

Cette technique est notamment utilisée pour réaliser *fork()*.



# Plan

---

- **Hiérarchie de mémoire**
- **Mémoire virtuelle**
  - ◆ Motivation et principes
  - ◆ Réalisation
  - ◆ Couplage de fichiers
- **Allocation dynamique de mémoire**
- **Pièges de la gestion mémoire en langage C**

# 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*).

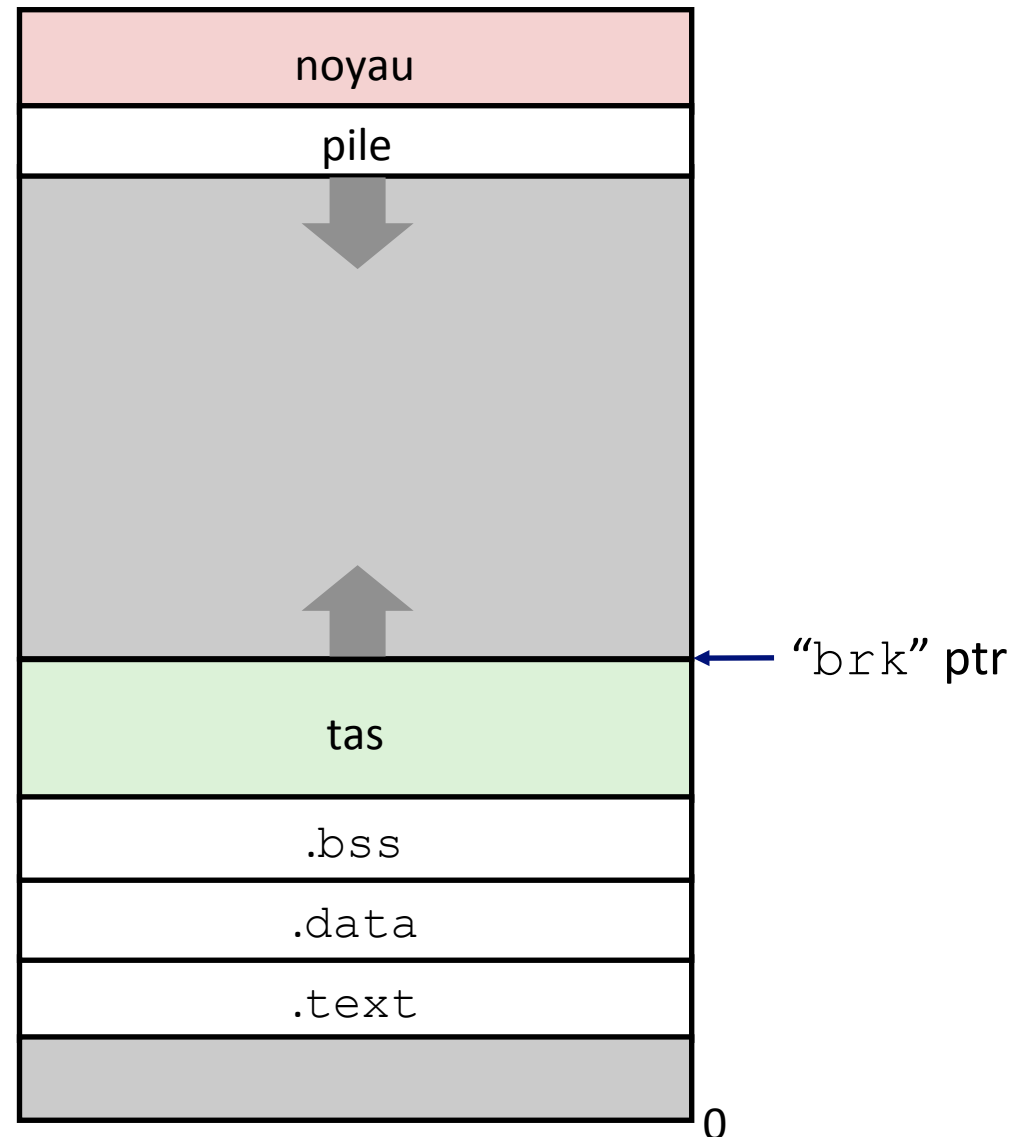
La pile et le tas occupent des segments différents dans la mémoire virtuelle d'un processus car leur mode de gestion est différent.



# 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 2<sup>nd</sup> edition*

# 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

## 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

# Système de gestion de la mémoire : Gestion explicite

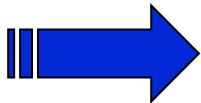
---

## ■ Rôle

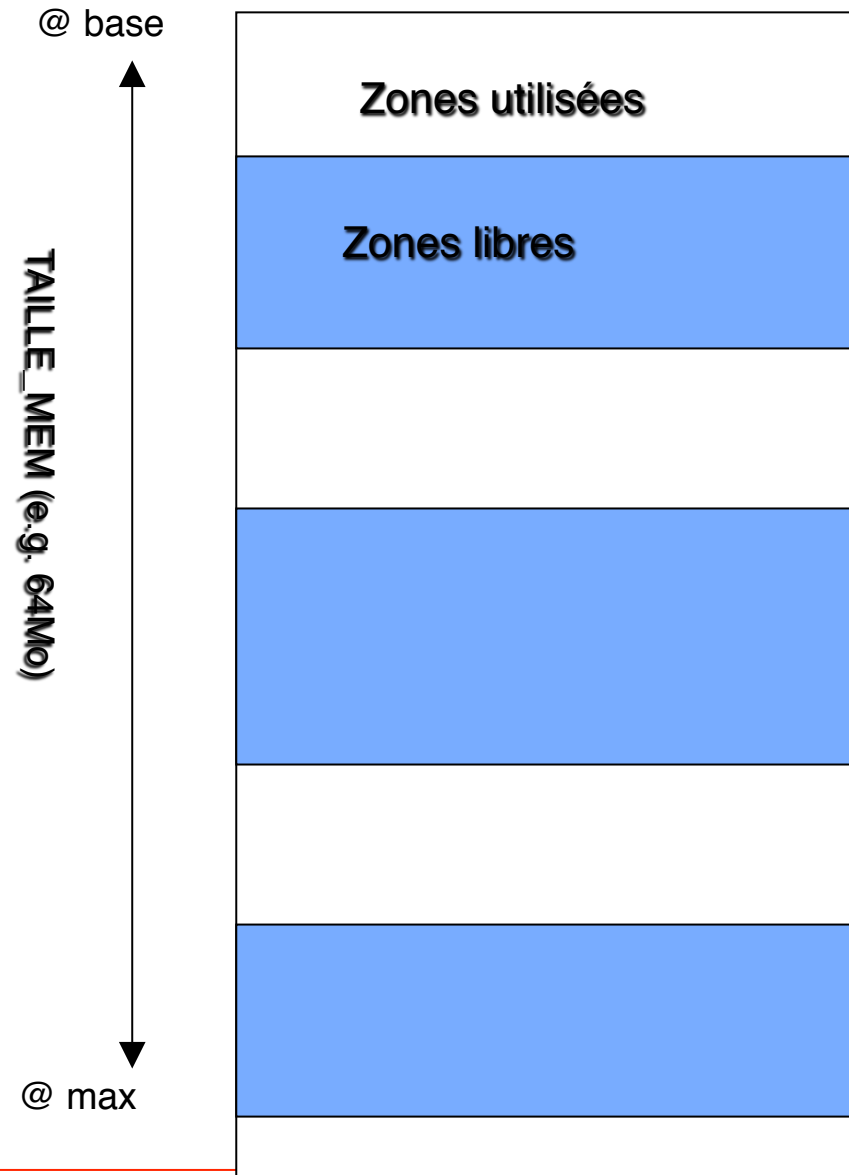
- ◆ Fournir des zones de mémoire aux programmes qui en demandent
- ◆ Gérer l'utilisation des zones mémoires disponibles

## ■ Deux types de mémoire

- ◆ Mémoire physique
  - ❖ Gestion de la mémoire matérielle de la plate-forme (e.g. RAM)
- ◆ Mémoire virtuelle
  - ❖ Fournir une plage mémoire plus large que celle disponible matériellement propre à chaque programme utilisateur



# Système de gestion de la mémoire physique



- **Connaître la mémoire physique**
- **Connaître les zones libres et les zones utilisées**
- **Fournir des zones libres lorsqu'un programme le demande**
  - ❖ Allocation
- **Libérer les zones 'rendues' par un programme**
  - ❖ Libération

# Utilisateurs

---

## ■ **Système d'exploitation**

- ◆ Utilisation pour son propre compte
- ◆ Gestion de la mémoire virtuelle

## ■ **Processus / Programmes utilisateurs**

- ◆ Allocation mémoire pour les structures propres à un programme

# Primitives d'Utilisation

---

## ■ Structures de descriptions de la mémoire et de son utilisation

- ❖ Description de la zone (@base, taille, ...)
- ❖ Descripteurs, tables des Zones libres et/ou utilisées

## ■ Initialisation

- ❖ Structures de gestion de la mémoire

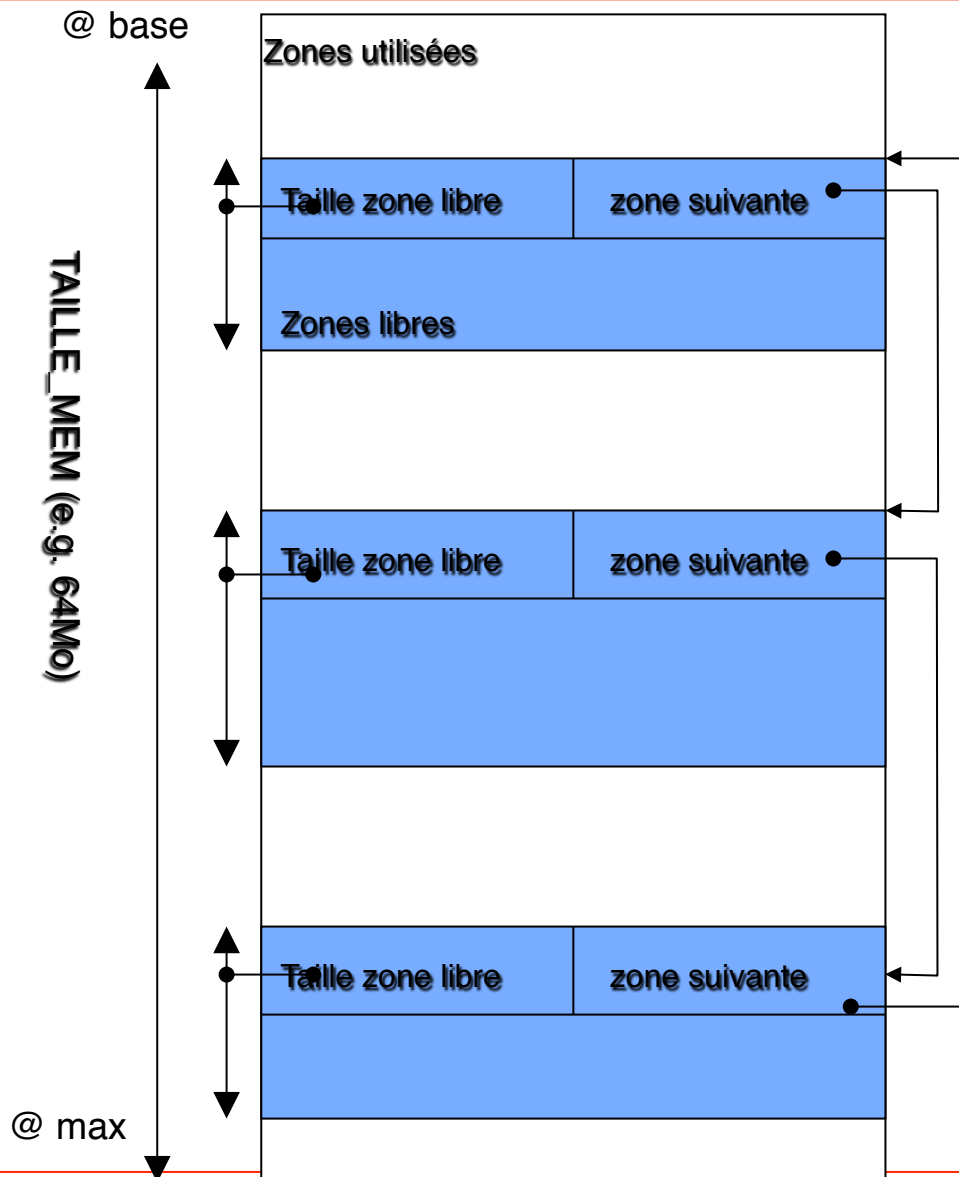
## ■ Allocation de mémoire

- ❖ retourne l'@ d'une zone libre de taille requise
- ❖ Allouer(taille)  $\succ$  pointeur de début de zone

## ■ Libération de mémoire

- ❖ Rend la zone précédemment allouée
- ❖ Libérer(@ zone, taille)  $\succ$  code d'erreur

# Algorithme de chaînage des zones libres



## ■ Informations sur les zone libres contenues dans les zones libres

- ❖ Taille de la zone libre
- ❖ @ de la zone libre suivante

## ■ Chaînage simple ou circulaire

## ■ Allocation

- ◆ Parcours de la liste des ZL
- ◆ Choix d'une ZL en fonction de la taille demandée
- ❖ Critères de choix variés: best fit, first fit, worst fit, ...

## ■ Libération

- ◆ Fusion éventuelle des zones libres adjacentes



# Avantages - Inconvénients

---

## ■ Avantages

- ◆ Utilisation des zones libres pour stocker les informations du SGM
- ◆ Algorithme simple

## ■ Inconvénients

- ◆ Performances
  - ❖ Parcours linéaire
- ◆ Fragmentation
  - ❖ Allocation de petite taille en général

# Algorithme du 'compagnon' (buddy system)

---

## ■ Allocation par blocs de tailles prédéfinies

❖ Blocs de  $2^k$ , pour une taille mémoire de  $2^{\max}$

## ■ Principe d'allocation

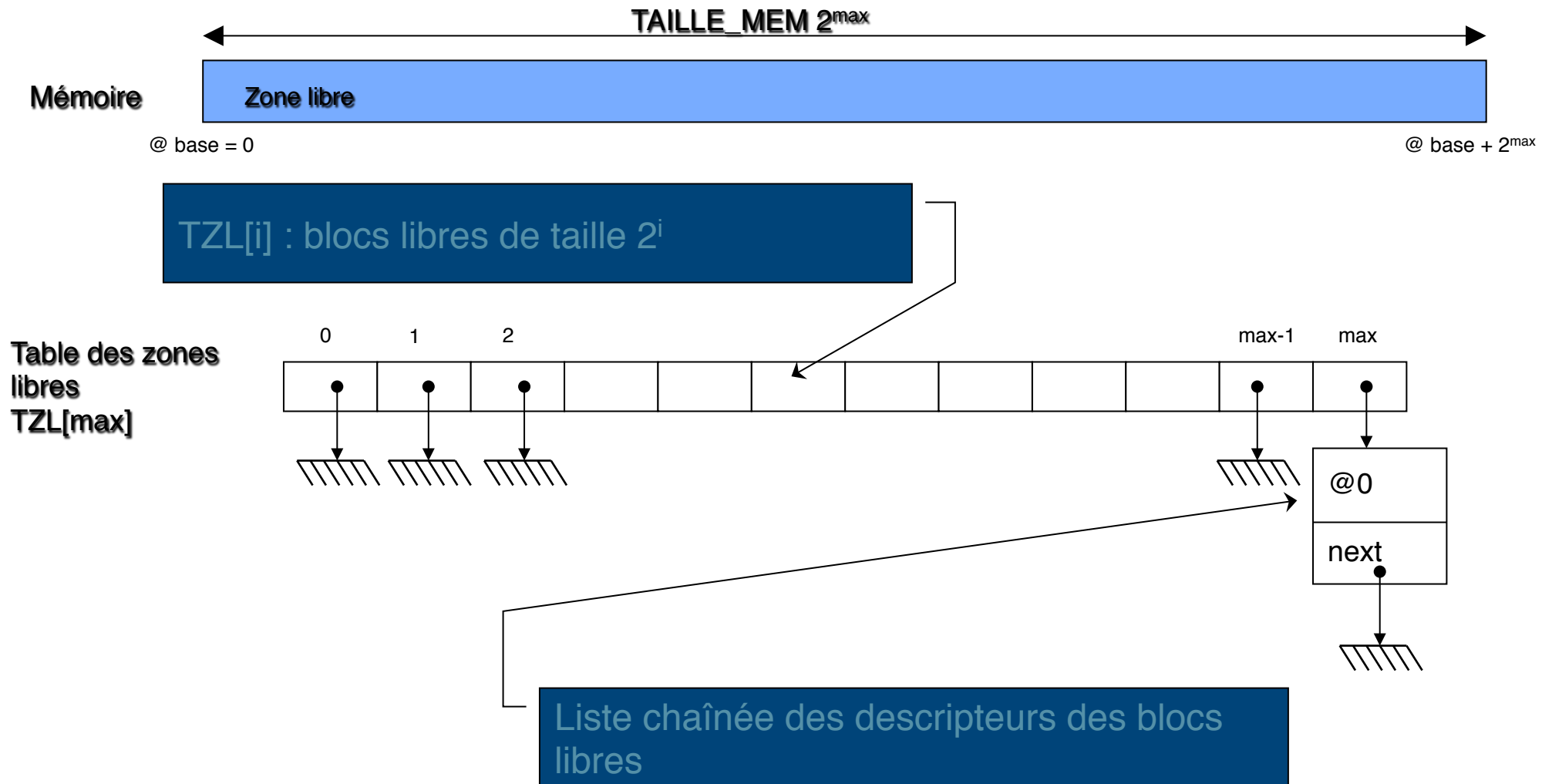
- ◆ Table des zones libres
- ◆ Recherche d'un bloc de taille  $2^k$
- ◆ Découpage des blocs libres en 2 blocs de taille inférieure (2 'compagnons') si possible

## ■ Principe de libération

- ◆ Recherche du 'compagnon' du bloc libéré
- ◆ Fusion des 'compagnons' si possible pour rendre un seul bloc libre de taille supérieur

# Allocation dans le 'buddy system'

État initial



# Allocation dans le 'buddy system'

Boucle d'allocation : bloc de taille  $p$  avec  $2^{k-1} < p < 2^k$

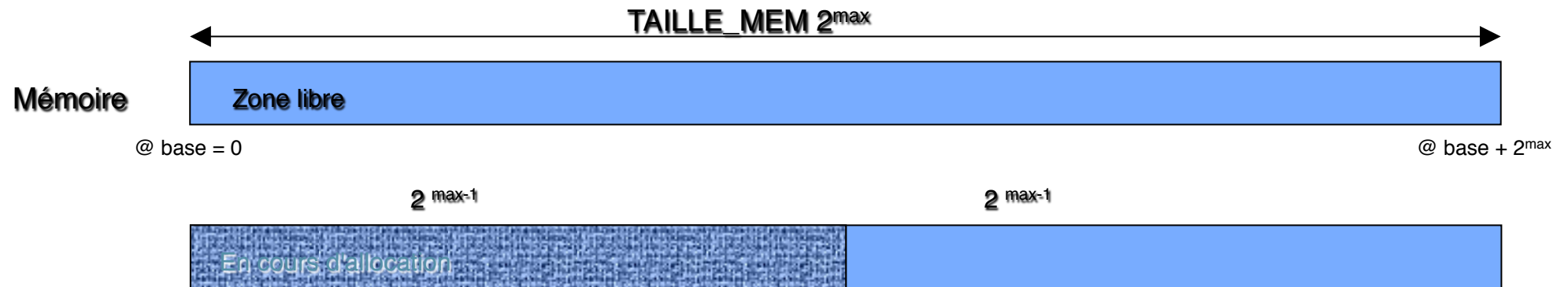
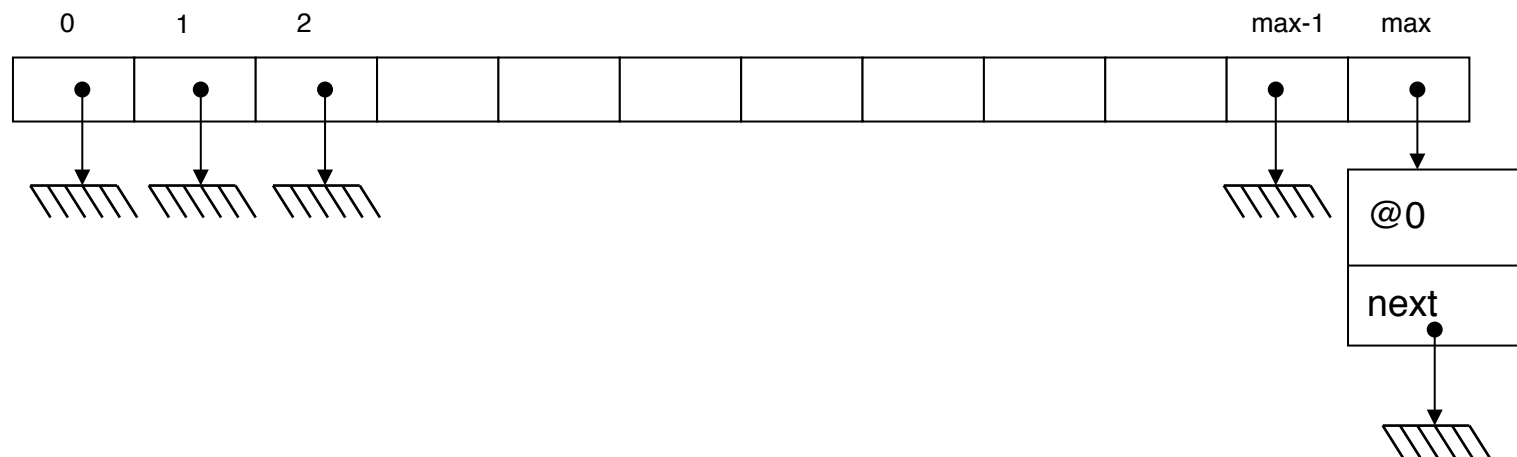


Table des zones  
libres  
TZL[max]



# Allocation dans le 'buddy system'

Boucle d'allocation : bloc de taille  $p$  avec  $2^{k-1} < p < 2^k$

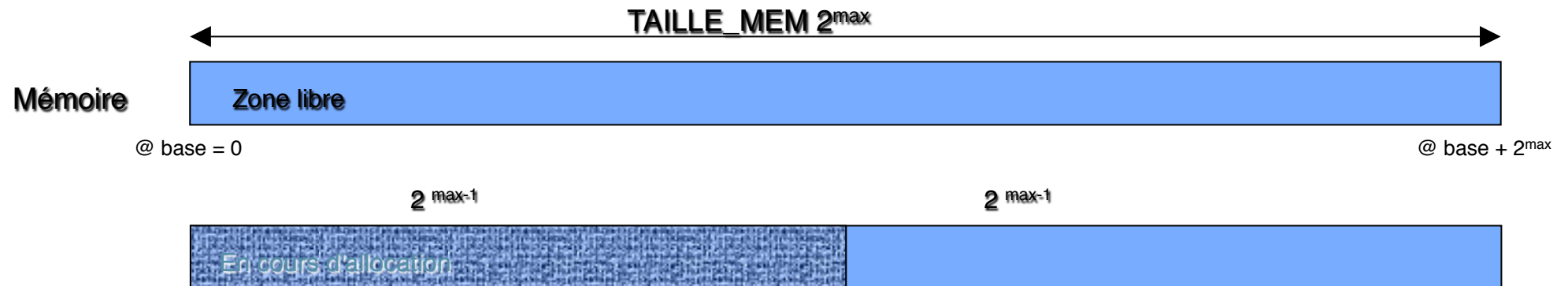
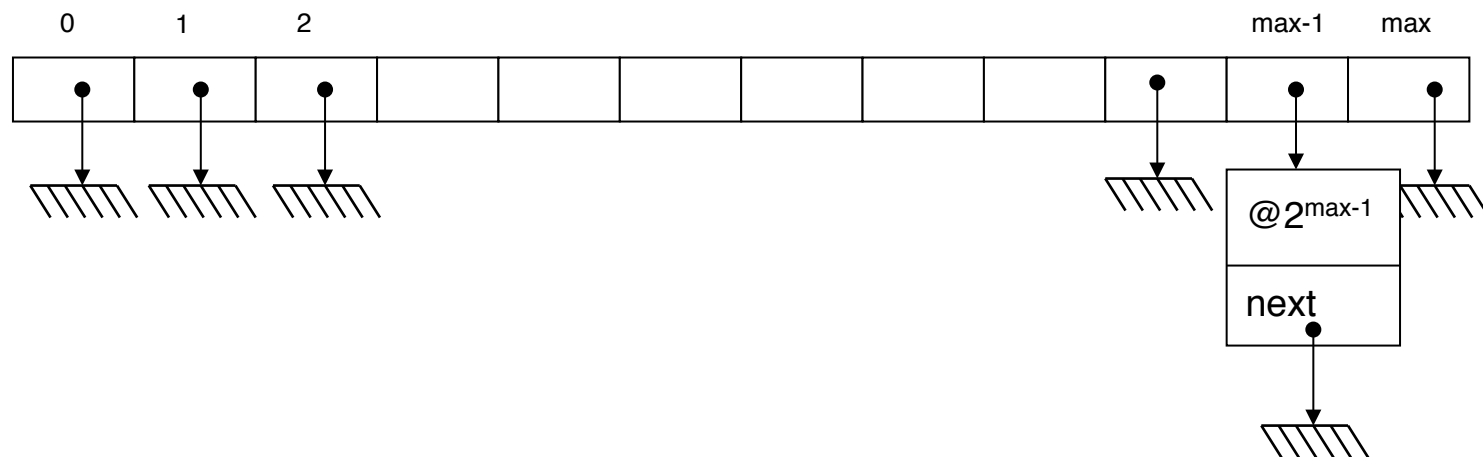
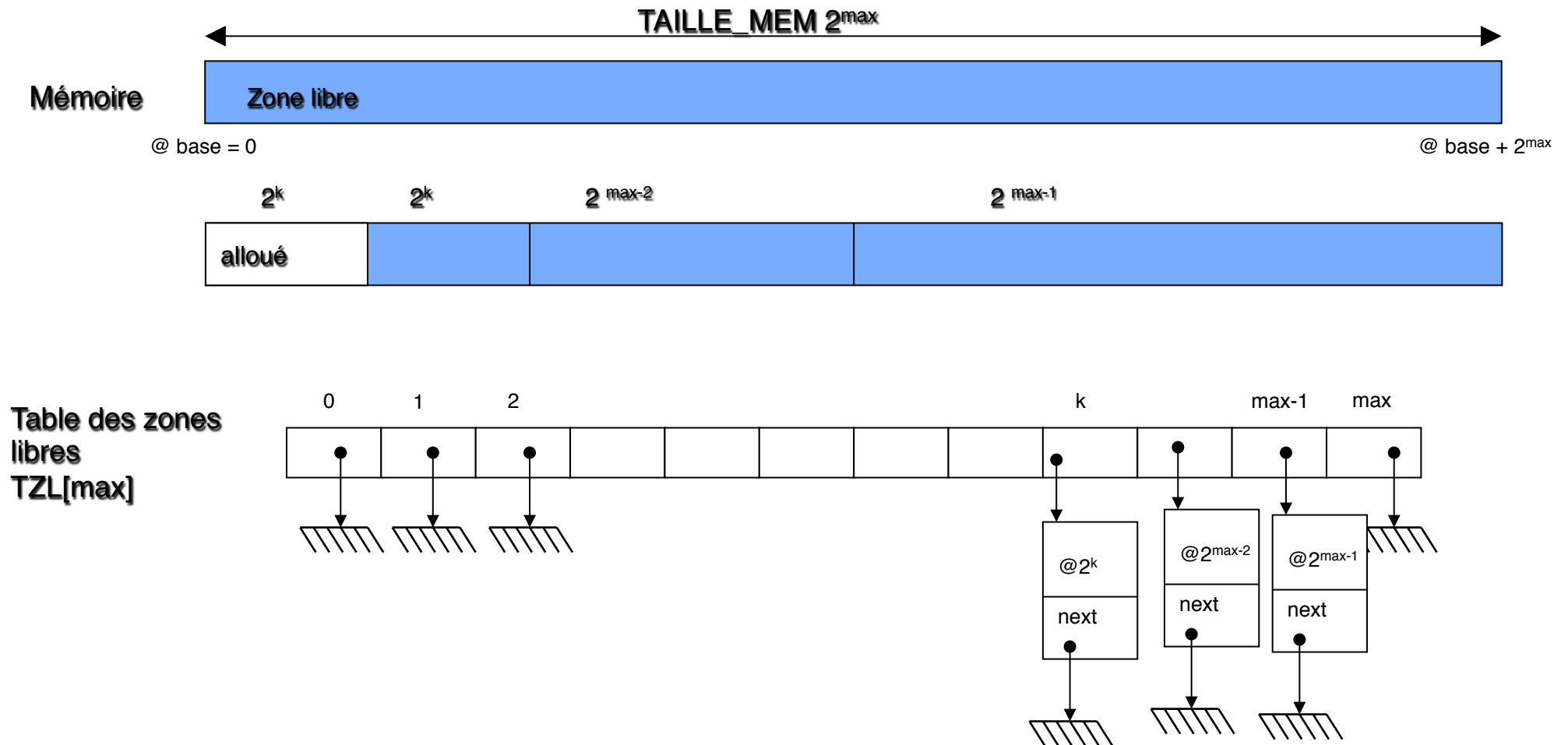


Table des zones  
libres  
TZL[max]



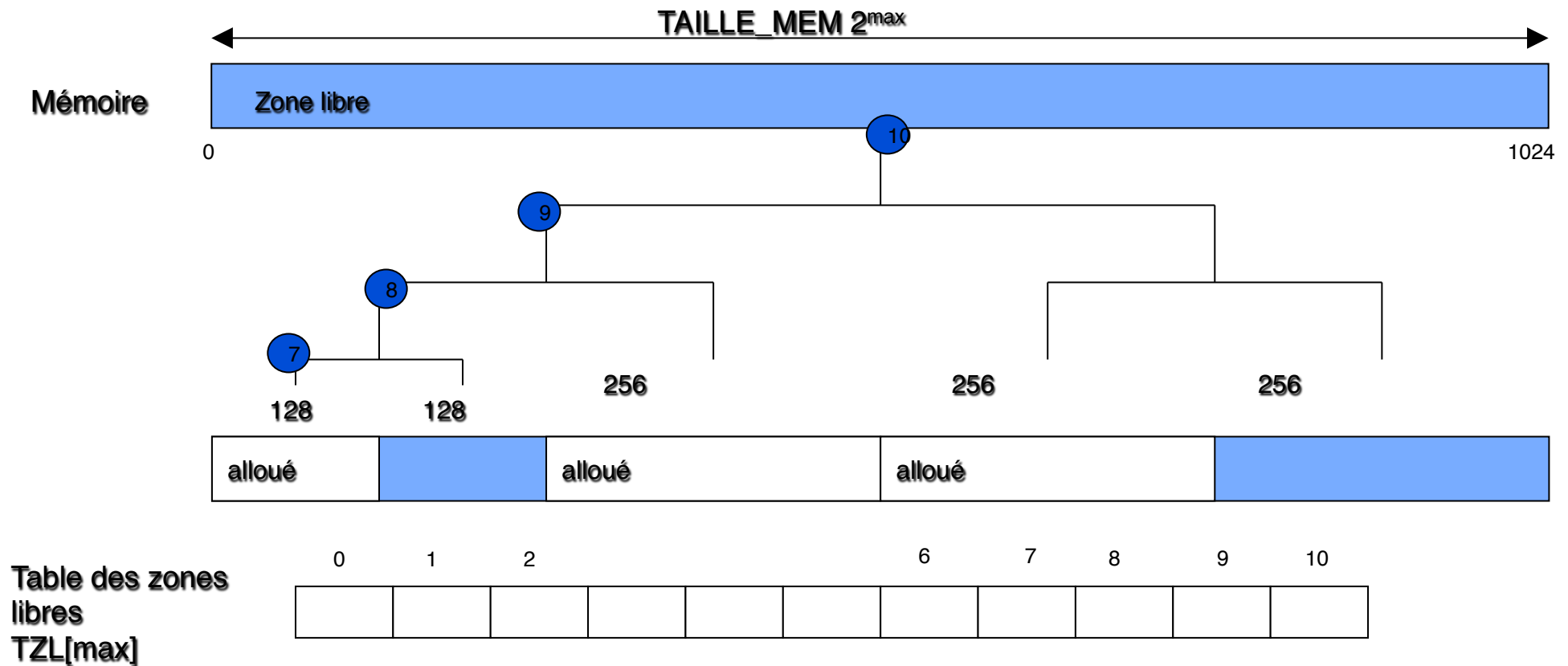
# Allocation dans le 'buddy system'

Boucle d'allocation : bloc de taille  $p$  avec  $2^{k-1} < p < 2^k$



# Allocation dans le 'buddy system'

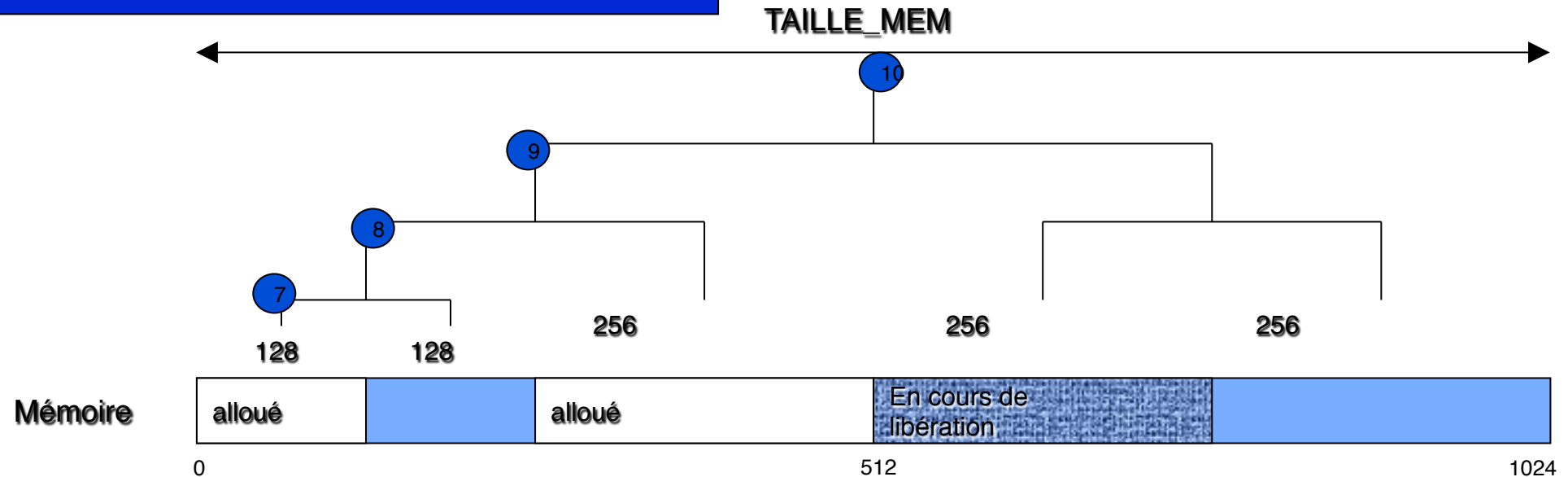
Exemple : taille 1024, blocs alloués



Remplir la table des zones libres

# Libération dans le 'buddy system'

Libération de @512, taille 256

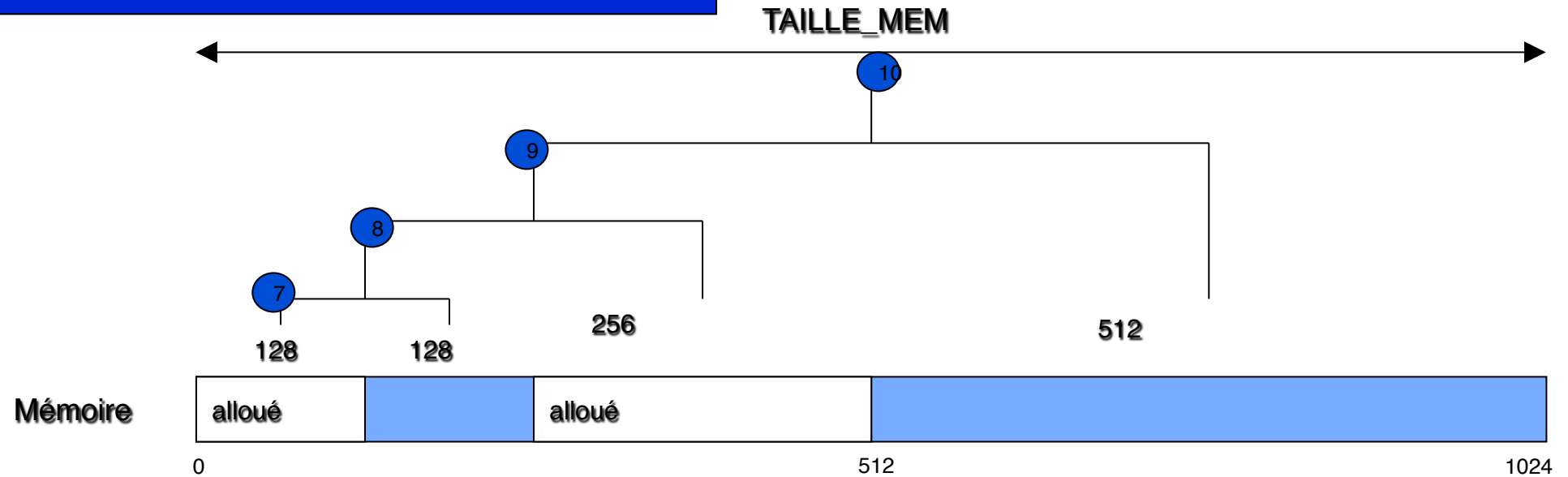


- Recherche de la zone à libérer
- Recherche du compagnon
- Fusion avec le compagnon s'il est libre
- Mise à jour de la table des zones libres



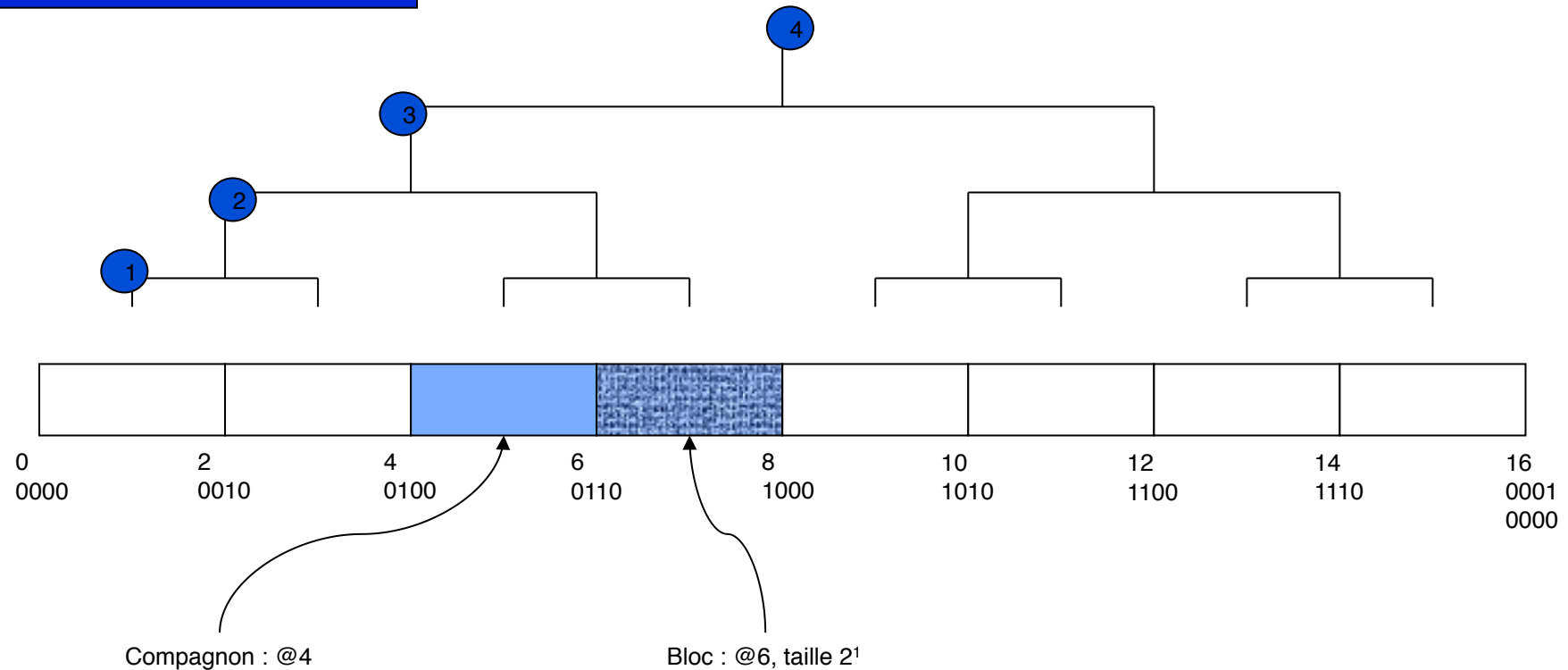
# Libération dans le 'buddy system'

Libération de @512, taille 256



# Recherche du compagnon

Mémoire de 16o



Algorithme efficace de calcul du compagnon

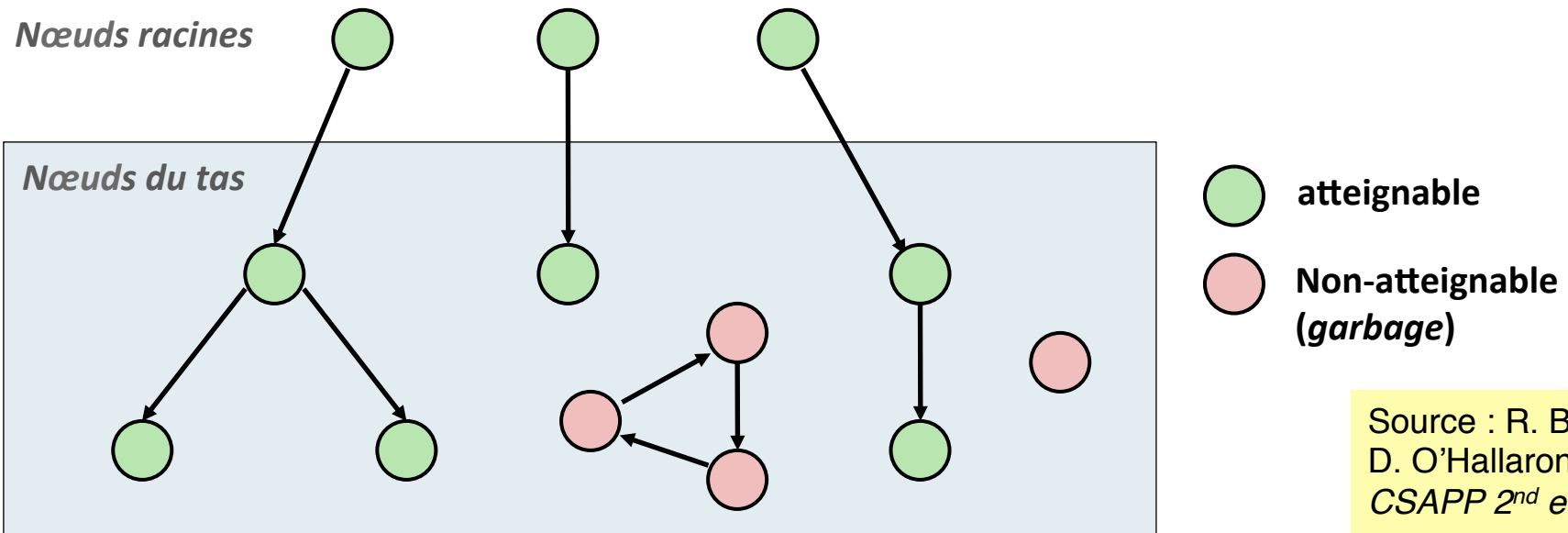
# Gestion mémoire implicite : Principe d'un ramasse-miettes (1)

---

- **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 » (*tracing garbage collectors*), qui est 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 2<sup>nd</sup> edition*

# Pièges classiques de la gestion mémoire en langage C (1)

---

```
int main () {  
    int val;  
    scanf("%d", val);  
    ...  
}
```

où est l'erreur ?

```
/* calcule A*X (matrice*vecteur) */  
  
int *matvec (int **A, int *X, int n) {  
    int *y = (int*) malloc(n*sizeof(int));  
    int i, j;  
    for (i=0; i<n; i++)  
        for (j=0; j<n; i++)  
            y[i] += A[i] [j] * X[j];  
    return y;  
}
```

où est l'erreur ?

Source : R. Bryant, D. O'Hallaron. *Computer Systems: a Programmer's Perspective*, Prentice-Hall, 2003

# Pièges classiques de la gestion mémoire en langage C (2)

```
/* créer un tableau de n par m */
int *makeArray (int n, int m) {
    int i ;
    int **A = (int **) malloc(n * sizeof(int));
    for (i=0; i<n; i++)
        A[i] = (int *) malloc(m * sizeof(int));
    return A;
}
```

On veut créer un tableau de  $n$  pointeurs dont chacun pointe vers un tableau de  $m$  ints

où est l'erreur ?

```
int main () {
    char buf [64];
    gets (buf);
    ...
}
```

où est l'erreur ?

Ne jamais utiliser gets() ; utiliser fgets()

Source : R. Bryant, D. O'Hallaron. *Computer Systems: a Programmer's Perspective*, Prentice-Hall, 2003

# Pièges classiques de la gestion mémoire en langage C (3)

---

```
/* créer un élément dans un tableau */  
int *search (int *p, int val) {  
    while (*p && *p != val)  
        p += sizeof(int);  
    return A;  
}
```

où est l'erreur ?

```
void fuite(int n) {  
    int *x = (int *) malloc(n * sizeof(int));  
    return;  
}
```

où est l'erreur ?

Source : R. Bryant, D. O'Hallaron. *Computer Systems: a Programmer's Perspective*, Prentice-Hall, 2003

# Pièges classiques de la gestion mémoire en langage C (4)

---

```
int *stackref() {  
    int val;  
    ...  
    return &val;  
}
```

où est l'erreur ?

```
int *stackref() {  
    int *x = (int *) alloca(n * sizeof(int));  
    ...  
    return x;  
}
```

où est l'erreur ?

Source : R. Bryant, D. O'Hallaron. *Computer Systems: a Programmer's Perspective*, Prentice-Hall, 2003



# Résumé de la séance 4

---

## ■ Motivations de la mémoire virtuelle

- ◆ Fenêtre sur l'ensemble des informations accessibles
- ◆ Espace propre à chaque processus (isolation)
- ◆ Protection des informations

## ■ Principes de réalisation de la mémoire virtuelle

- ◆ Introduction à la mémoire virtuelle paginée

## ■ Utilisation de la mémoire virtuelle par les processus

- ◆ Compléments sur *fork()* et *exec()* ; copie sur écriture
- ◆ Couplage de fichiers : *mmap()*

## ■ Allocation dynamique de mémoire

- ◆ Primitives
- ◆ Pièges de l'allocation dynamique de mémoire