

INF2610

Noyau d'un système d'exploitation



Chapitre 7 - Gestion de la mémoire

Sommaire



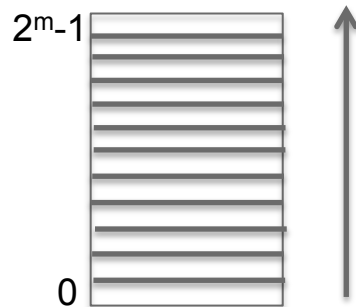
- Gestion de la mémoire physique
 - Espace d'adressage physique
 - Politiques d'allocation d'espace
- Gestion de la mémoire virtuelle
 - Pagination pure
 - Segmentation sans/avec pagination
- Gestion de l'espace d'adressage d'un processus (cas de Linux et C)

Gestion de la mémoire physique

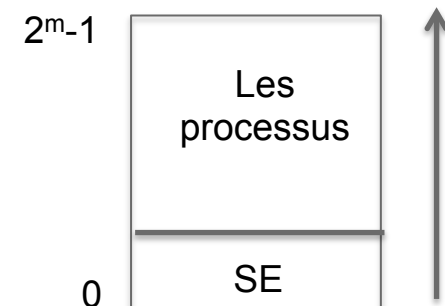
- Le gestionnaire de la mémoire est le composant du système d'exploitation qui se charge de gérer l'allocation de l'espace physique nécessaire à l'exécution du système d'exploitation (SE) et des processus.
- La multiprogrammation impose la cohabitation de plusieurs processus en mémoire physique.
- Comment gérer efficacement l'allocation de cet espace mémoire partagé ?
 - Organisation de la mémoire physique.
 - Politique de placement.
 - Allocation statique avec/sans va-et-vient ou à la demande avec/sans pré-allocation.
 - Politique de remplacement.
- ➔ utiliser efficacement les processeurs et les périphériques tout en évitant l'écroulement du système et assurant la protection des espaces privés des processus.

Espace d'adressage physique

- La mémoire physique (principale) est l'espace de travail des processeurs. Toute instruction ou donnée d'un processus doit être chargée en mémoire physique pour qu'elle soit accessible aux processeurs.
- Elle est vue par chaque processeur comme un tableau de mots de 1 octet (8 bits) chacun. Chaque mot est référencé par une adresse physique.
- Un adressage physique sur m bits est suffisant pour référencer les 2^m octets de la mémoire physique.

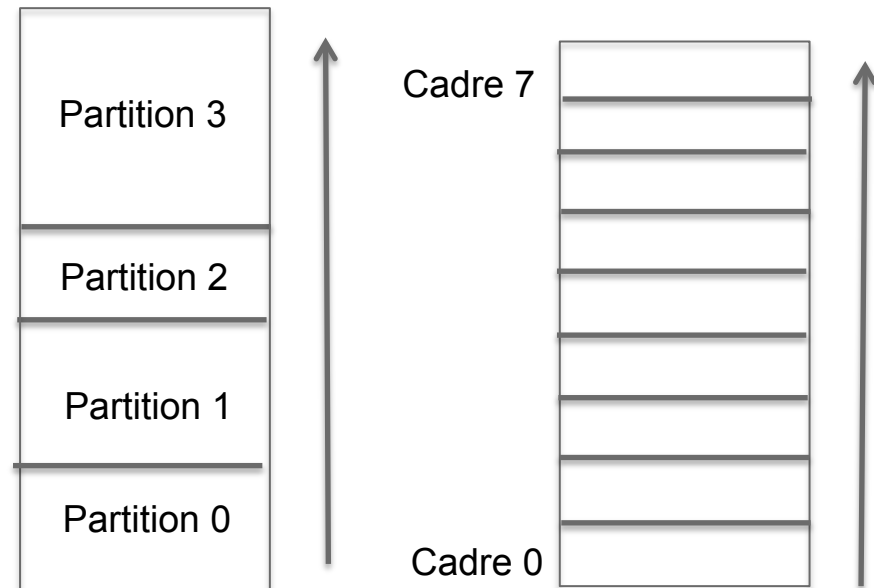


- Cet espace est généralement partitionné en 2 parties : une réservée au système d'exploitation (accessible en mode noyau) et une autre réservée aux processus.



Espaces d'adressage physique

- L'unité d'allocation aux processus est :
 - l'octet,
 - une partition, si la mémoire physique est partitionnée en zones de tailles différentes, ou
 - le cadre, si la mémoire physique est partitionnée en cadres de même taille en général de 4 KiO,



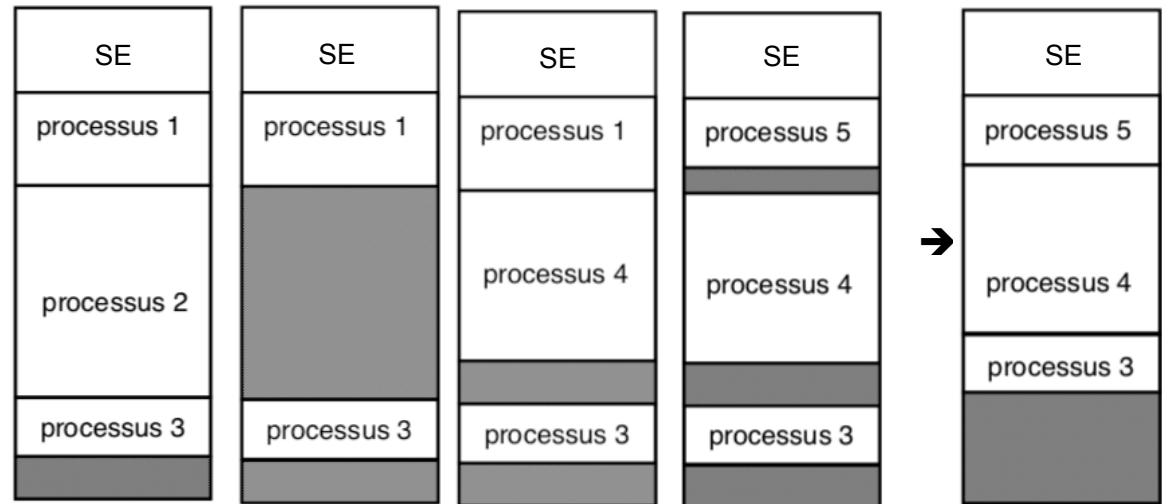
Comment gérer les demandes d'allocation d'espace ?

Politiques d'allocation d'espace

- L'allocation et la libération d'espaces contigus (de tailles différentes) dynamiquement finissent par créer des trous de tailles différentes.
- Parmi ces trous, lequel est alloué en premier ?
 - premier ajustement (First-fit),
 - meilleur ajustement (Best-fit),
 - pire ajustement (Worst-fit), ou
 - par subdivision (Buddy system)

→ **Fragmentation externe**
(espaces libres trop petits
entre des espaces alloués).

→ **Compactage coûteux**



Premier ajustement (First-fit)

Compactage

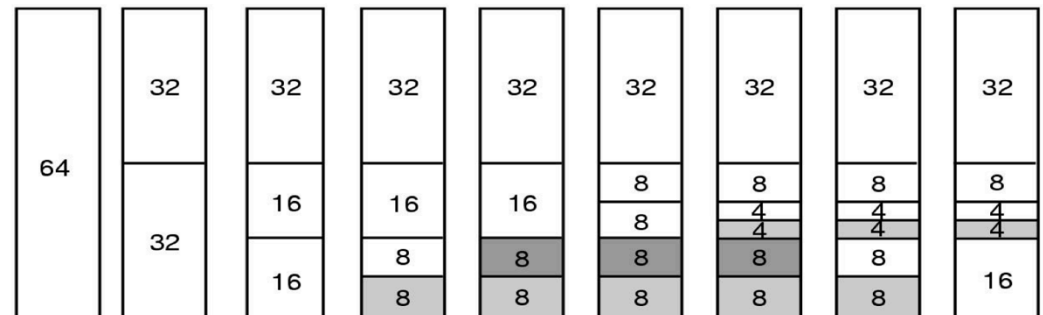
Politiques d'allocation d'espace

Allocation par subdivision (utilisée par Linux pour allouer des espaces physiques contigus) :

- Initialement, la mémoire physique est composée d'une seule zone vide.
 - Lorsqu'une allocation est demandée, la taille de l'espace demandé x est arrondi à la puissance de 2 la plus proche (y).
 - Ensuite, l'allocateur recherche la plus petite zone vide de taille supérieure ou égale à y .
 - Si la taille de la zone est y alors cette zone est allouée.
 - Sinon la zone est divisée en 2 parties égales et le même processus de division est répété sur l'une des deux parties.
-
- Dans le cas de Linux, la mémoire physique est structurée en cadres de même taille (4KiO) et l'unité d'allocation est le cadre.

Politiques d'allocation d'espace

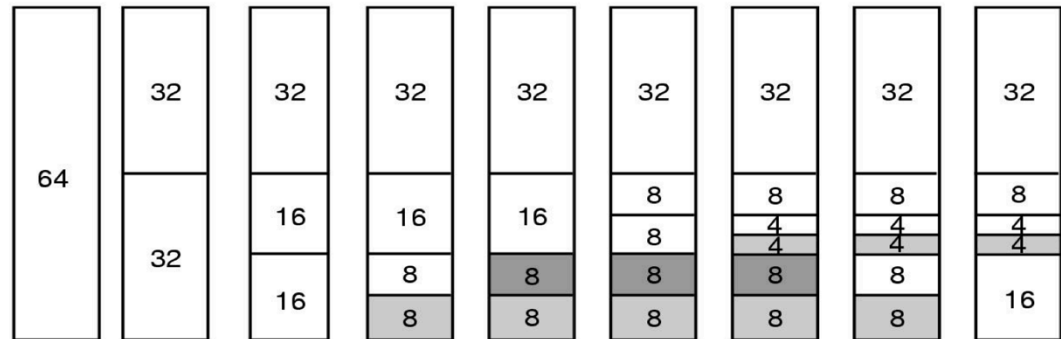
Allocation par subdivision



- On demande 6 cadres et on dispose d'une zone de 64 cadres libres :
 - On arrondit à 8 cadres.
 - $64 > 8 \rightarrow$ On divise la zone de 64 cadres en 2 zones de 32 cadres chacune.
 - $32 > 8 \rightarrow$ On divise une zone de 32 en 2 zones de 16 cadres chacune.
 - $16 > 8 \rightarrow$ On divise une zone de 16 en 2 zones de 8 cadres chacune.
 - $8 = 8 \rightarrow$ On alloue une zone de 8.
- Lors de la libération d'espace, les zones libres adjacentes de même taille sont regroupées. Ainsi, les tailles de toutes les zones libres sont des puissances de 2.

Politiques d'allocation d'espace

Allocation par subdivision



- Pour gérer les espaces libres en mémoire physique, Linux utilise un tableau `free_area` de pointeurs sur des listes doublement chaînées. L'entrée `free_area[i]` pointe sur la liste doublement chaînée de zones libres de taille 2^i cadres.
- ➔ Fragmentations externe et interne (espace alloué mais non utilisé).

Comment gérer le manque d'espace mémoire ?

- ➔ Technique de va-et-vient (retrait temporaire des processus de la mémoire).
- ➔ Mémoire virtuelle : allocation de la mémoire à la demande durant l'exécution.

Gestion de la mémoire virtuelle

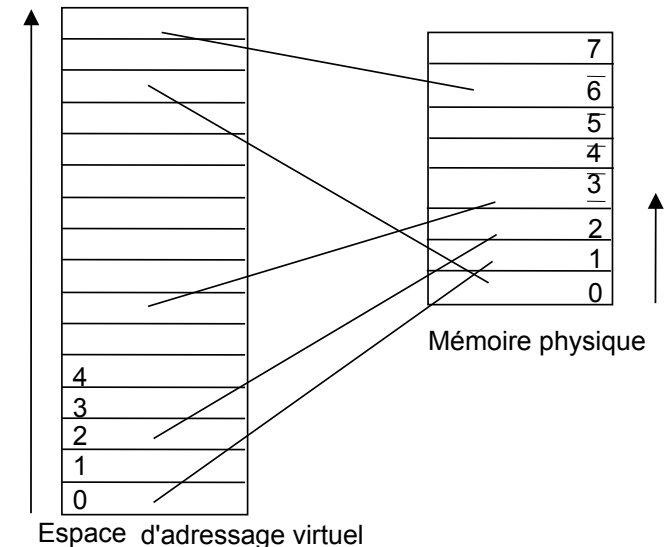
- On distingue trois principales techniques de gestion de la mémoire virtuelle :
 - pagination pure,
 - segmentation pure et
 - segmentation combinée avec pagination.
- Elles diffèrent principalement par l'organisation des espaces virtuels et espace physique et la translation d'adresses.

Pagination pure

- L'espace d'adressage d'un processus (espace virtuel) est un espace linéaire organisé en pages de même taille.
- Le processus peut référencer chaque mot de cet espace par une adresse relative à cet espace (adresse virtuelle ou logique).
- Les adresses figurant dans les instructions sont virtuelles (c-à-d elles font référence à cet espace).
- Elles sont traduites en adresses physiques durant l'exécution par le processeur exécuteur.

Pagination pure

- La mémoire physique est aussi un espace linéaire composé de cadres de même taille.
- La taille d'une page est égale à celle d'un cadre.
- L'unité de chargement et d'allocation de la mémoire physique à un processus est le cadre
→ fragmentation interne (espace alloué mais non utilisé).
- Les pages de l'espace d'adressage sont chargées dans les cadres de la mémoire physique à la demande (pas nécessairement dans un espace contigu).



**int getpagesize();
(4 KiO = 4096 octets).**

Pagination pure - Format des adresses

- D'une manière générale, pour un espace de 2^X octets, l'adressage est sur X bits. Chaque code binaire sur X bits référence un octet de cet espace.
- Si cet espace est partitionnée en pages/cadres de 2^Y octets chacune (avec $2^Y < 2^X$) alors :
 - cet espace est composé de 2^{X-Y} pages/cadres et
 - dans chaque adresse sur X bits, les $(X-Y)$ bits de poids le plus fort donne le numéro de pages et les Y bits restants donne le déplacement dans la page.

Numéro de page

Déplacement

Pagination pure - Format des adresses

Numéro de page

Déplacement

Exemple 1 : Supposez un espace d'adressage de 64 KiO et une taille de page de 4 KiO.

- L'espace d'adressage est composé $64 \text{ KiO} / 4 \text{ KiO/page} = 16 \text{ pages}$.
 - L'adresse virtuelle est sur 16 bits (car $64 \text{ KiO} = 2^{16} \text{ octets}$).
 - Le déplacement est sur 12 bits (car $4 \text{ KiO} = 2^{12} \text{ octets}$).
- ➔ L'adresse virtuelle sur 16 bits est composée de :
- (16-12) bits (de poids le plus fort) qui indique le numéro de page et
 - 12 bits (de poids le plus faible) qui donne le déplacement dans la page.

Pagination pure - Format des adresses

Numéro de page	Déplacement
----------------	-------------

Exemple 1 : Supposez un espace d'adressage de 64 KiO et une taille de page de 4 KiO.

Donnez le numéro de page ainsi que le déplacement dans cette page de l'octet 20000 de cet espace.

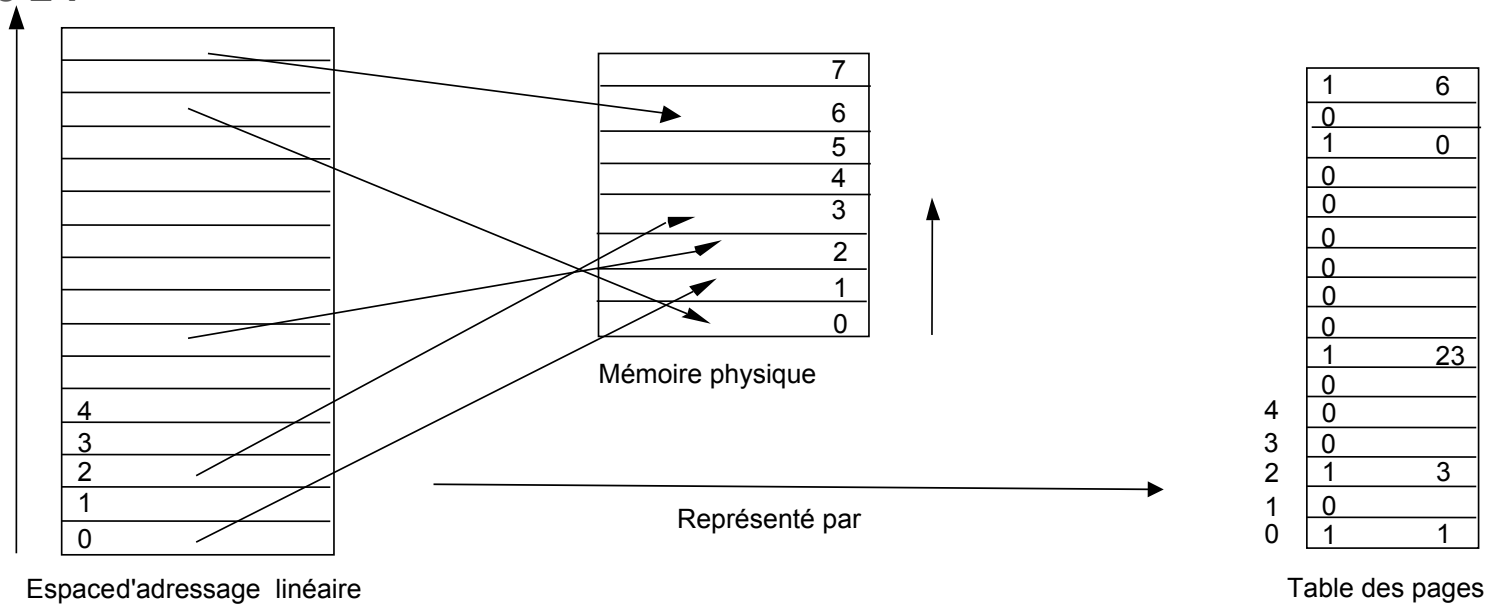
- L'octet 20000 de cet espace est l'octet **$20000 \% 4096 = 3616$** de la page **$20000 / 4096 = 4$** .
- Si cette page est chargée dans le cadre 5 de la mémoire physique, l'octet 20000 de l'espace virtuel est chargé dans l'octet 3616 du cadre 5 ce qui correspond à l'octet $5 * 4096 + 3616$ de la mémoire physique.

Pagination pure - Table des pages

- Au niveau du système, l'espace d'adressage virtuel d'un processus est représenté par une table de pages (TDP) qui contient autant d'entrées qu'il y a de pages dans l'espace d'adressage virtuel du processus.
- Chaque entrée est composée de :
 - **un bit de présence en mémoire physique,**
 - un bit de référence,
 - bits de protection,
 - un bit de modification,
 - **un numéro de cadre,**
 - un emplacement sur le disque ou dans la zone de swap, etc.
- TDP sert à localiser les pages de l'espace d'adressage virtuel et à traduire les adresses virtuelles en adresses physiques → Elle doit être chargée en mémoire principale, à des fins de performances, lorsque le processus est en exécution.

Pagination pure – Table des pages

Exemple 2 :



- Les pages 0, 2, 6, 13 et 15 du processus sont en mémoire dans les cadres 1, 3, 2, 0 et 6.
- Les autres pages sont supposés sur le disque ou dans la zone de swap.

Pagination pure – Table des pages

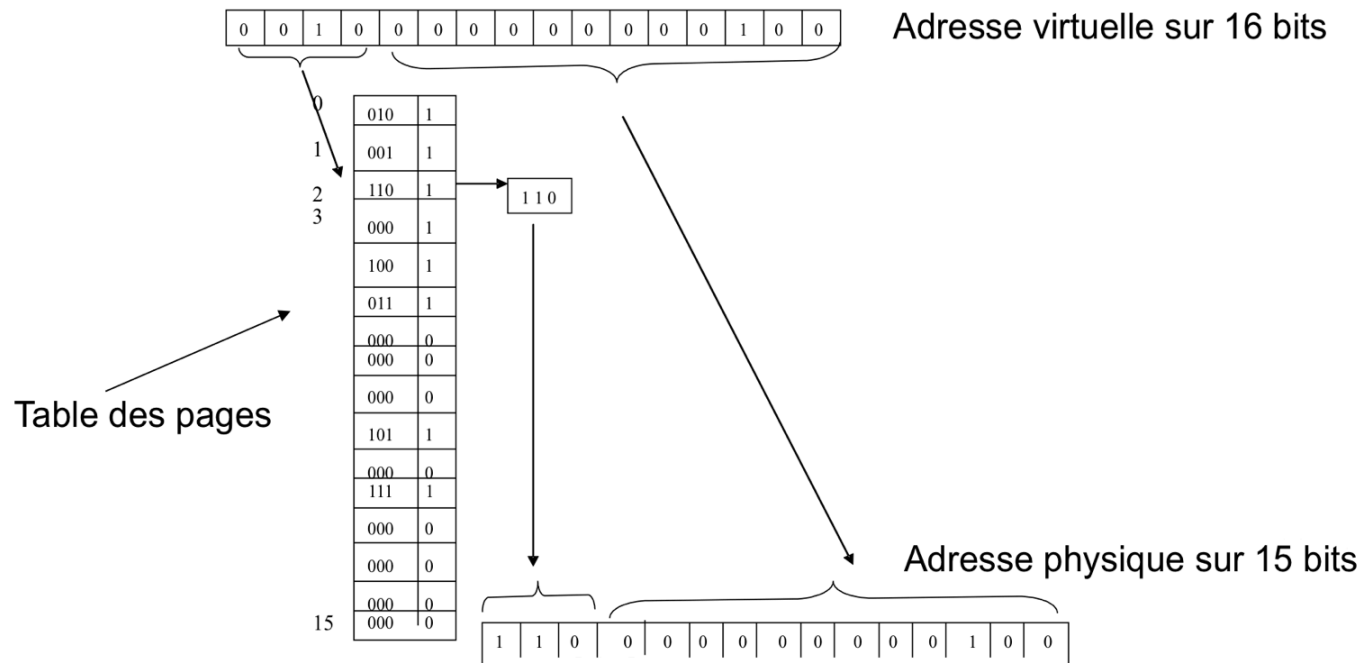
Exemple 2 : Traduire l'adresse virtuelle (2, 5).

Table des pages				
Index	Bit de présence	Bit de référence	...	Numéro de cadre
0	1	0	...	1
1	0	0	...	0
2	1	1	...	3

L'adresse physique recherchée est une adresse dont le numéro de cadre est 3 et l'offset est 5

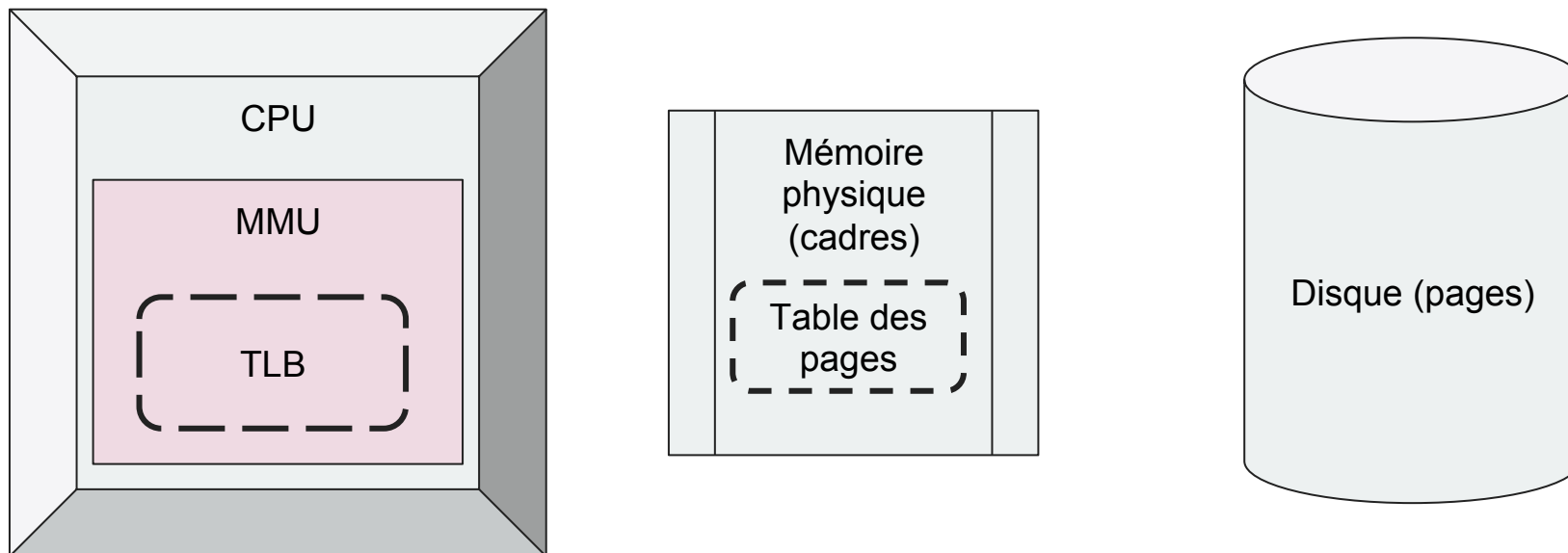
Pagination pure – Table des pages

Exemple 3 : Supposez une machine de 64 KiO (2^{16} octets) de mémoire virtuelle, 32 KiO (2^{15} octets) de mémoire physique et des pages de 4 KiO (2^{12} octets).



Pagination pure – Translation d'adresses

- La conversion d'adresses est réalisée par un composant matériel du processeur appelé MMU (Memory Management Unit). Ce dernier est doté d'une **mémoire associative appelée TLB** (Translation Lookaside Buffer).



Pagination pure - Translation d'adresses

- Le TLB agit comme un cache. Il mémorise les informations sur les **dernières pages référencées par le processus en cours d'exécution**.
- Chaque entrée contient :
 - Un bit de validité,
 - Un numéro de page virtuelle,
 - Un bit de modification,
 - des bits de protection et
 - Un numéro de cadre.
- Les bits de validité sont remis à 0 lors d'un changement de contexte.

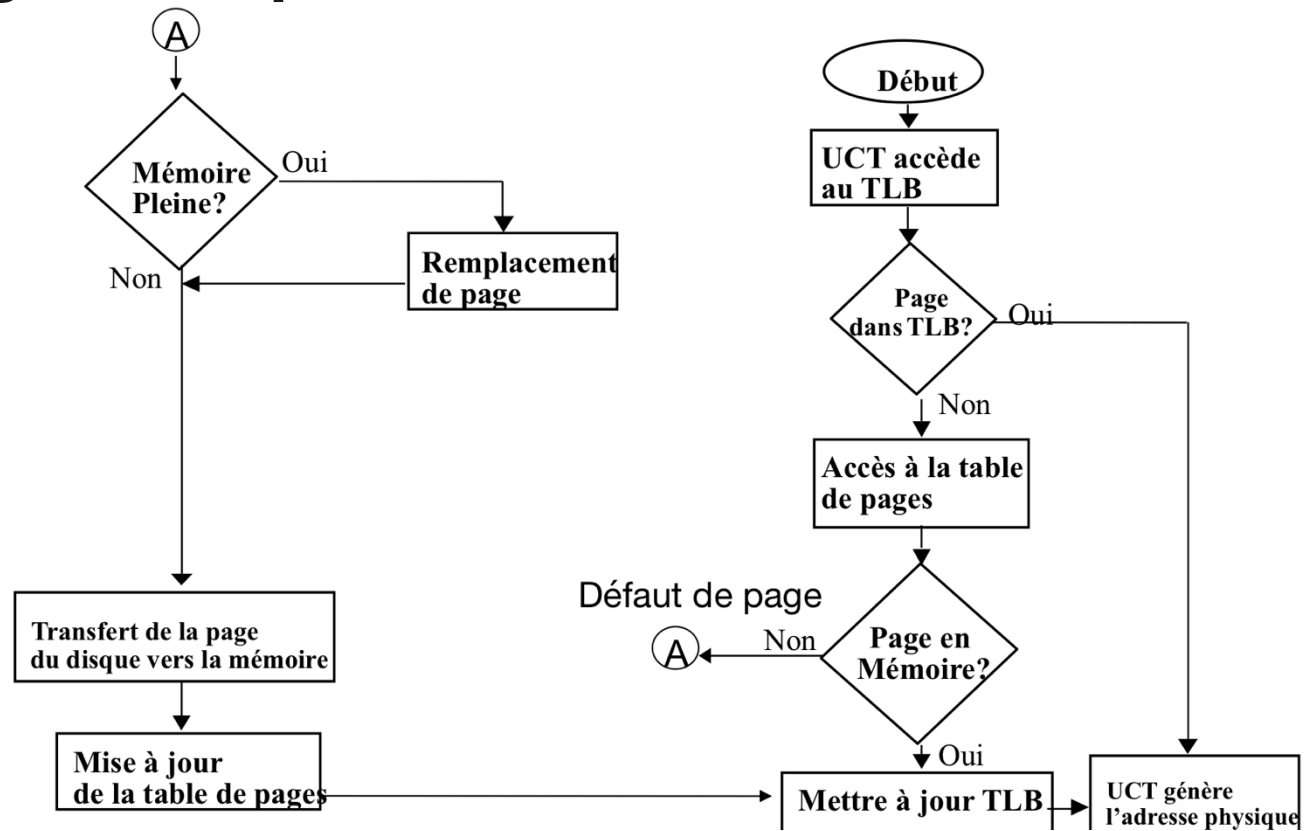
Pagination pure - Translation d'adresses

- Lorsqu'une adresse virtuelle (p,d) doit être traduite en adresse physique, le MMU vérifie d'abord si le numéro de page p est présent dans le TLB et le bit de validité correspondant est à 1.
- Si c'est le cas et les bits de protection sont conformes au mode d'accès, le MMU récupère le numéro de cadre c figurant dans cette entrée puis génère l'adresse physique en remplaçant le numéro de page p par le numéro de cadre c. Si les bits de protection ne sont pas conformes au mode d'accès, **un défaut de protection se produit.**
- Si p n'est pas présent dans le TLB, le MMU **poursuit sa recherche dans la table des pages.**

Pagination pure - Translation d'adresses

- Si le **bit de présence** à l'entrée p de la table des pages du processus est à 1 et les bits de protection sont conformes au mode d'accès, le MMU récupère le numéro de cadre c puis génère l'adresse physique en remplaçant le numéro de page p par le numéro de cadre c. Si les bits de protection ne sont pas conformes au mode d'accès, **un défaut de protection se produit**.
- Si le **bit de présence** à l'entrée p de la table des pages est à 0, le MMU génère un **défaut de page** pour localiser la page p et éventuellement lancer son chargement en mémoire physique si la page est sur le disque (**un défaut de page dur**). Dans ce cas, le processus passe à l'état bloqué. Lorsque la page p est chargée en mémoire, la table des pages est mise à jour et le processus redevient prêt. Il s'agit d'un **défaut de page léger**, si la page est en mémoire physique mais non associée au processus ou encore elle n'est pas en mémoire physique et est de type « **page zéro** ».

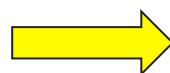
Pagination pure - Translation d'adresses



Pagination pure – Translation d'adresses

Exemple 4 : Traduction de l'adresse virtuelle (4,12) :

État du TLB: La page 4 est présente dans la TLB.




Bit de validité	Bit de modification	Bits de protection	Numéro de page	Numéro de cadre
1	0	...	1	4
1	0	...	0	3
1	0	...	4	2

L'adresse physique de (4,12) est : (2,12)

Pagination pure - Translation d'adresses

Exemple 5 : Traduction de l'adresse virtuelle (3,20) :

État du TLB (remplacement FIFO) :




Bit de validité	Bit de modification	Bits de protection	Numéro de page	Numéro de cadre
1	0	...	1	4
1	0	...	0	3
1	0	...	4	2

Table des pages				
Index	Bit de présence	Bit de référence	...	Numéro de cadre
0	1	1	...	3
1	1	1	...	4
2	0	0	...	0
3	1	0	...	6
4	1	1		2

Pagination pure - Translation d'adresses

Exemple 5 : Traduction de l'adresse virtuelle (3,20) :

État du TLB (remplacement FIFO) :



Bit de validité	Bit de modification	Bits de protection	Numéro de page	Numéro de cadre
1	0	...	4 3	4 6
1	0	...	0	3
1	0	...	4	2


L'adresse physique de (3,20) : (6, 20)

Table des pages				
Index	Bit de présence	Bit de référence	...	Numéro de cadre
0	1	1	...	3
1	1	1	...	4
2	0	0	...	0
3	1	0	...	6
4	1	1		2

Pagination pure - Translation d'adresses

Exemple 6 : Traduction de l'adresse virtuelle (3,20) :

État du TLB (remplacement FIFO) :



Bit de validité	Bit de modification	Bits de protection	Numéro de page	Numéro de cadre
1	0	...	1	4
1	0	...	0	3
1	0	...	4	2


- Défaut de page => Chargement de la page 3 dans, par exemple, le cadre 8.

Table des pages				
Index	Bit de présence	Bit de référence	...	Numéro de cadre
0	1	1	...	3
1	1	1	...	4
2	0	0	...	0
3	0	0	...	0
4	1	1		2

Pagination pure - Translation d'adresses

Exemple 6 : Traduction de l'adresse virtuelle (3,20) :

État du TLB (remplacement FIFO) :



Bit de validité	Bit de modification	Bits de protection	Numéro de page	Numéro de cadre
1	0	...	4 3	4 8
1	0	...	0	3
1	0	...	4	2

- Défaut de page => Chargement de la page 3 dans, par exemple, le cadre 8.
- L'adresse physique de (3,20) est (8,20).

Table des pages				
Index	Bit de présence	Bit de référence	...	Numéro de cadre
0	1	1	...	3
1	1	1	...	4
2	0	0	...	0
3	0 1	0 1	...	0 8
4	1	1		2

Pagination pure – Politiques de remplacement

- Lorsqu'un défaut de page survient, le système doit charger la page manquante en mémoire physique.
- **Si la mémoire est pleine**, le système doit **retirer une page en mémoire pour la remplacer par la page manquante**.
- Parmi les pages en mémoire physique, laquelle nous devons retirer en premier, afin de **minimiser le nombre de défauts de page et éviter l'écroulement du système ?**
- Le choix de la page à retirer peut **se limiter aux pages du processus ayant causé le défaut de page (allocation locale)** ou à **l'ensemble des pages en mémoire (allocation globale)**.
- En général, l'allocation **globale** produit de meilleurs résultats.

Pagination pure – Politiques de remplacement

Algorithme optimal de Belady :

Critère: Remplacer la page qui sera référéncée le plus tard possible dans le FUTUR

- Non réalisable mais utilisé à des fins de comparaison.
- Version allocation locale et globale.

Exemple : On a un système avec 3 cadres et un processus qui référence dans cet ordre les pages de son espace d'adressage virtuel :

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Défaut de page!
7 est la page qui
sera référencée le
plus tard!

Algorithme optimal (Belady)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

[illegible]

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

[illegible]

Pagination pure – Politiques de remplacement



Algorithme optimal (Belady)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	2	2	2	2	2								
	0	0	0	0	0	0	4	4	4	0	0								
		1	1	1	3	3	3	3	3	3	3								

Pagination pure – Politiques de remplacement



Algorithme optimal (Belady)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	2	2	2	2	2	2							
	0	0	0	0	0	0	4	4	4	0	0	0							
		1	1	1	3	3	3	3	3	3	3	3							

Pagination pure – Politiques de remplacement



Défaut de page!
3 est la page qui
sera référencée le
plus tard!

Algorithme optimal (Belady)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	2	2	2	2	2	2							
	0	0	0	0	0	0	4	4	4	0	0	0	0						
		1	1	1	3	3	3	3	3	3	3	3	1						

Pagination pure – Politiques de remplacement



Algorithme optimal (Belady)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	2	2	2	2	2	2	2	2					
	0	0	0	0	0	0	4	4	4	0	0	0	0	0					
		1	1	1	3	3	3	3	3	3	3	3	1	1					

Pagination pure – Politiques de remplacement

Algorithme optimal (Belady)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2				
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0				
		1	1	1	3	3	3	3	3	3	3	3	1	1	1				

Pagination pure – Politiques de remplacement

Algorithme optimal (Belady)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2			
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0			
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1		

Pagination pure – Politiques de remplacement



Défaut de page!
2 est la page qui
sera référencée le
plus tard!

Algorithme optimal (Belady)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7		
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0		
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1		

Pagination pure – Politiques de remplacement

Algorithme optimal (Belady)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	

Pagination pure – Politiques de remplacement

Algorithme optimal (Belady)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1

En résumé :

- Nombre d'accès: 20
- Défauts de page: 9 (45%)

Pagination pure – Politiques de remplacement

Algorithme FIFO

Critère : remplacer la page dont le temps de résidence est le plus long.

- Implémentation facile
- Pas basé sur le principe de localité (une page référencée a tendance à être réutilisée pendant une certaine période).
- **Anomalie de Belady**: le nombre de défauts de page pourrait augmenter avec le nombre de cadres.

Exemple : On a un système avec 3 cadres et les références suivantes :

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Défaut de page!
2 est le first in!

Algorithme FIFO

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

[illegible]

Défaut de page!
0 est le first in!

Algorithme FIFO

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

[illegible]

Pagination pure – Politiques de remplacement



Algorithme FIFO

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	4	4	4	0	0								
	0	0	0	0	3	3	3	2	2	2	2								
		1	1	1	1	0	0	0	3	3	3								

Pagination pure – Politiques de remplacement



Algorithme FIFO

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	4	4	4	0	0	0							
	0	0	0	0	3	3	3	2	2	2	2	2							
		1	1	1	1	0	0	0	3	3	3	3							

Pagination pure – Politiques de remplacement



Défaut de page!
2 est le first in!

Algorithme FIFO

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	4	4	4	0	0	0	0						
	0	0	0	0	3	3	3	2	2	2	2	2	1						
		1	1	1	1	0	0	0	3	3	3	3	3						

Pagination pure – Politiques de remplacement



Défaut de page!
3 est le first in!

Algorithme FIFO

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	4	4	4	0	0	0	0	0					
	0	0	0	0	3	3	3	2	2	2	2	2	1	1					
		1	1	1	1	0	0	0	3	3	3	3	3	2					

Pagination pure – Politiques de remplacement



Algorithme FIFO

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0				
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1				
		1	1	1	1	0	0	0	3	3	3	3	3	2	2				

Pagination pure – Politiques de remplacement



Algorithme FIFO

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0			
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1			
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2			

Pagination pure – Politiques de remplacement



Défaut de page!
0 est le first in!

Algorithme FIFO

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7		
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1		
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2		

Pagination pure – Politiques de remplacement



Défaut de page!
1 est le first in!

Algorithme FIFO

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	

Pagination pure – Politiques de remplacement



Défaut de page!
2 est le first in!

Algorithme FIFO

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1

En résumé :

- Nombre d'accès: 20
- Défauts de page: 15 (75%)

Pagination pure – Politiques de remplacement

Nombre d'accès: 20
Nombre de défauts de page: 16 (80%)

Algorithme FIFO - Anomalie de belady

Si pour le même exemple, le nombre de cadres 4 au lieu de 3 :

7	0	1	2	7	0	3	7	0	1	2	3	7	0	1	2	3	2	4	3
7	7	7	7	7	7	3	3	3	3	2	2	2	2	1	1	1	1	1	1
	0	0	0	0	0	0	7	7	7	7	3	3	3	3	2	2	2	2	2
		1	1	1	1	1	1	0	0	0	0	7	7	7	7	3	3	3	3
			2	2	2	2	2	2	1	1	1	1	0	0	0	0	0	4	4

Pagination pure – Politiques de remplacement

Algorithme LRU

Critère : remplacer la page la moins récemment utilisée.

- Basé sur le principe de localité.
- Difficile à implémenter (approximations implémentées dans UNIX et Linux).

Exemple : On a un système avec 3 cadres et les références suivantes :

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Pagination pure – Politiques de remplacement

Algorithme LRU

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	4	4	4	0	0								
	0	0	0	0	0	0	0	0	3	3	3								
		1	1	1	3	3	3	2	2	2	2								

Pagination pure – Politiques de remplacement

Algorithme LRU

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	4	4	4	0	0	0							
	0	0	0	0	0	0	0	0	3	3	3	3							
		1	1	1	3	3	3	2	2	2	2	2							

Pagination pure – Politiques de remplacement



Défaut de page!
0 est LRU!

Algorithme LRU

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	4	4	4	0	0	0	1						
	0	0	0	0	0	0	0	0	3	3	3	3	3						
		1	1	1	3	3	3	2	2	2	2	2	2						

Pagination pure – Politiques de remplacement

Algorithme LRU

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	4	4	4	0	0	0	1	1					
	0	0	0	0	0	0	0	0	3	3	3	3	3	3					
		1	1	1	3	3	3	2	2	2	2	2	2	2					

Pagination pure – Politiques de remplacement



Défaut de page!
3 est LRU!

Algorithme LRU

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1				
	0	0	0	0	0	0	0	0	3	3	3	3	3	3	0				
		1	1	1	3	3	3	2	2	2	2	2	2	2	2				

Pagination pure – Politiques de remplacement

Algorithme LRU

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
		1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7

En résumé :

- Nombre d'accès: 20
- Défauts de page: 12 (60%)

Pagination pure – Politiques de remplacement

Algorithme de l'horloge (seconde chance)

Approximation de l'algorithme LRU

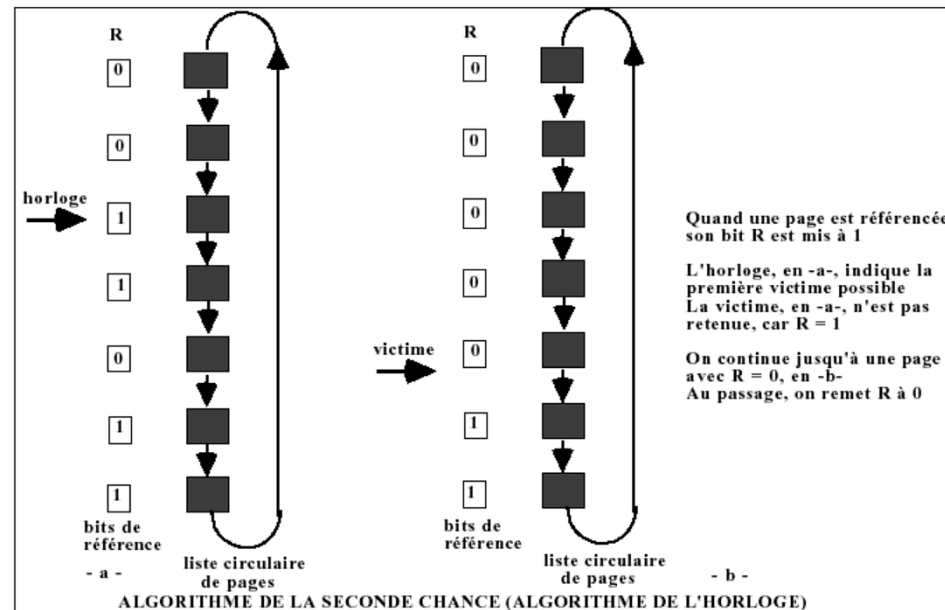
- Les pages en mémoire sont enregistrées dans une liste circulaire (horloge)
- On place un indicateur sur la **page la plus ancienne**,
- Un bit de référence R est associé à chaque page
- Dès qu'une page est référencée, son bit R est mis à 1
- Lorsqu'on veut retirer une page, on parcourt la liste circulaire, en commençant par celle pointée par l'indicateur, à la recherche d'une page avec R=0.
- Le bit R des pages consultées est mis à 0.
- La première page rencontrée ayant un bit R à 0 est remplacée par la nouvelle page. Le bit R de la page ajoutée est initialisé à 1

➔ **Une variante de cet algorithme tient compte du bit de modification M.**

Pagination pure – Politiques de remplacement

Algorithme de l'horloge (seconde chance)

Algorithme de l'horloge (seconde chance)



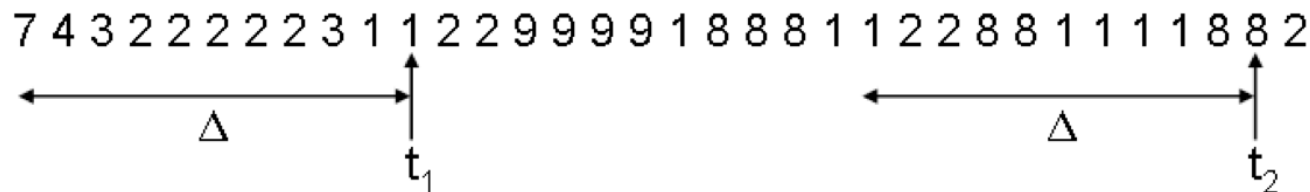
Pagination pure – Politiques de remplacement

Espace de travail (working set)

(Windows <https://docs.microsoft.com/en-us/windows/win32/memory/working-set?redirectedfrom=MSDN>)

- $WS(t, \Delta)$ est l'ensemble des pages ayant été référencées entre les instants $t-\Delta$ et t
 - $WS(t, \Delta)$ ne doit pas excéder une certaine limite.
- ➔ Retirer une page en mémoire qui n'est pas dans l'espace de travail.

Exemple: pour $\Delta = 10$, $WS(t_1, \Delta) = \{1, 2, 3, 4, 7\}$ et $WS(t_2, \Delta) = \{1, 2, 8\}$



https://fr.wikipedia.org/wiki/M%C3%A9moire_virtuelle#Le_working_set:_espace_de_travail

Pagination pure – Politiques de remplacement



Cas de LINUX

Au démarrage du système, le processus init crée un processus démon de pages (kswapd) qui exécute un algorithme de remplacement de pages (proche de l'algorithme de l'horloge).

Ce démon est réveillé **périodiquement** ou **à la suite de l'allocation d'un grand espace mémoire** pour vérifier si le nombre de cadres disponibles est trop bas.

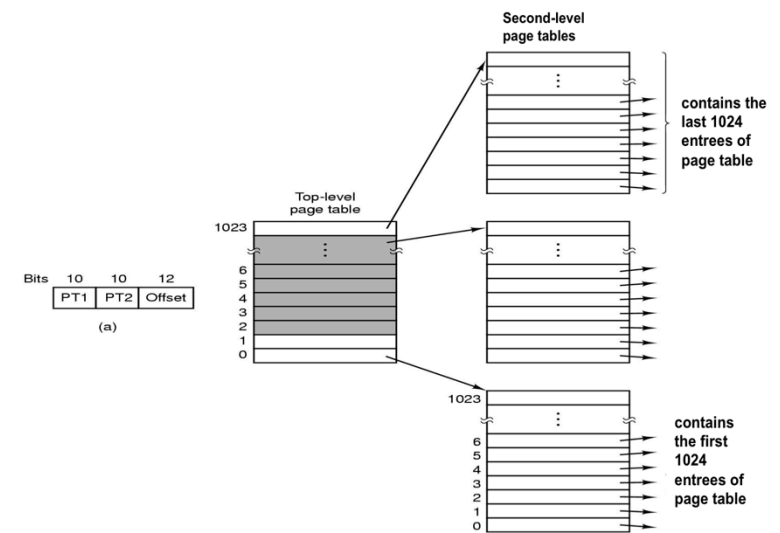
- Si c'est le cas, il recherche les pages (inactives et actives) à libérer.
- Si ce n'est pas le cas, il se rendort.

Pagination pure - Table des pages à plusieurs niveaux

- La table des pages contient autant d'entrées qu'il y a de pages pour un processus donné.
- La taille de la table des pages peut ainsi être très grande : 2^{20} entrées pour un espace d'adressage virtuel de 32 bits et des pages de 4 KiO.
- Afin d'éviter d'avoir des pages trop grandes en mémoire, des tables de pages à plusieurs niveaux sont utilisées.
- Par exemple, la table des pages de 2^{20} entrées peut être partitionnée en 1024 tables de 1024 entrées que l'on pourrait charger séparément.
- Il faut une table qui contiendra les pointeurs vers chacune des 1024 tables. Cette table est dite du premier niveau. Les 1024 tables sont des tables du second niveau.

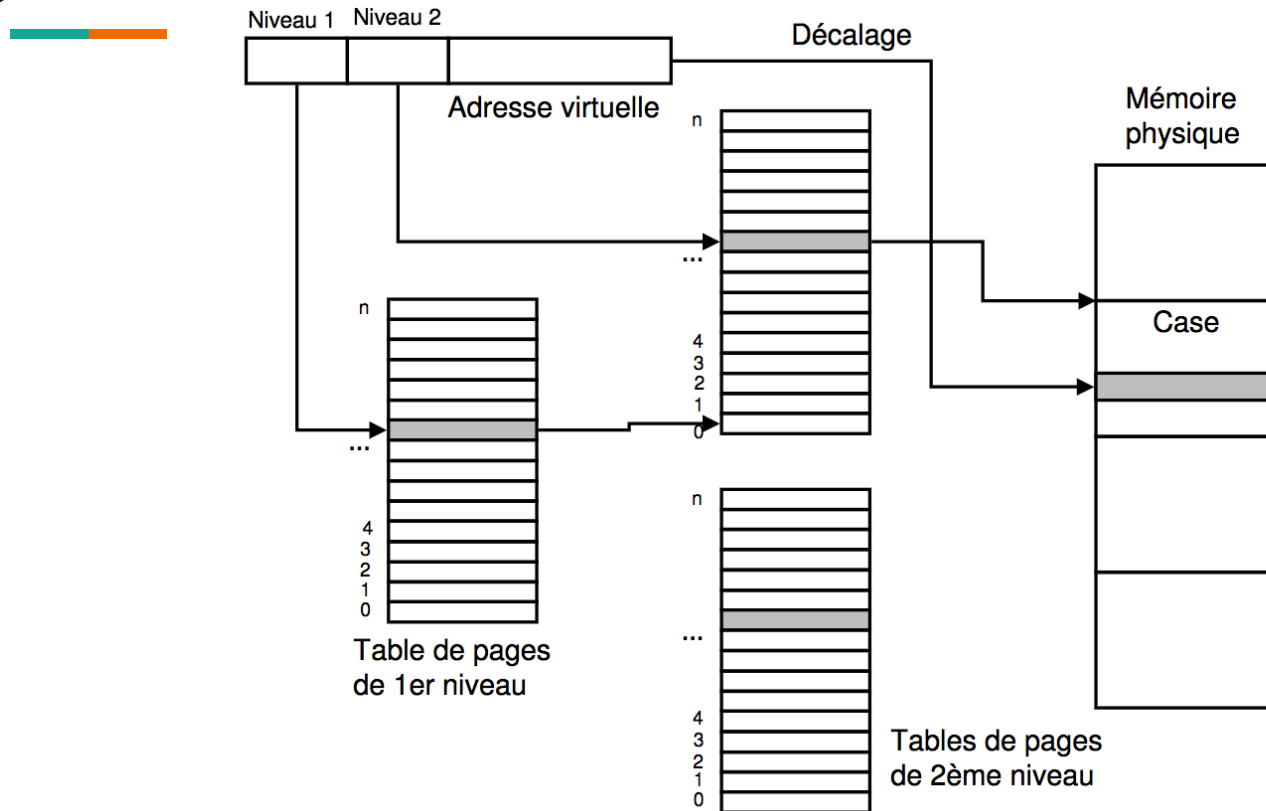
Pagination pure - Table des pages à plusieurs niveaux

- Reprenons l'exemple de notre table des pages de 2^{20} entrées que nous avons partitionnée en une table d'index (premier niveau) et 1024 tables de 1024 entrées (second niveau).
- Le format de l'adresse est :
 - 10 bits pour identifier l'entrée dans la table d'index (table de pages du premier niveau),
 - 10 bits pour identifier l'entrée dans la table de pages du deuxième niveau et
 - 12 bits pour le déplacement.

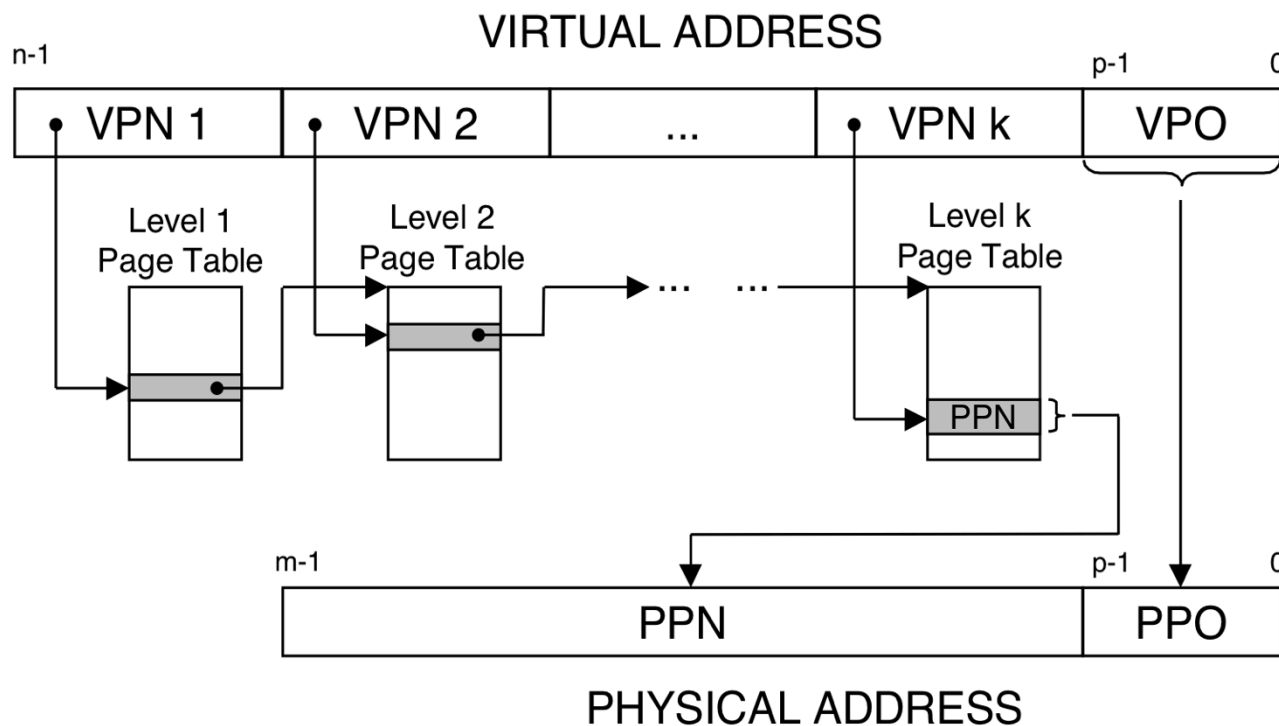


Tannenbaum

Pagination pure - Table des pages à plusieurs niveaux



Pagination pure - Table des pages à K niveaux

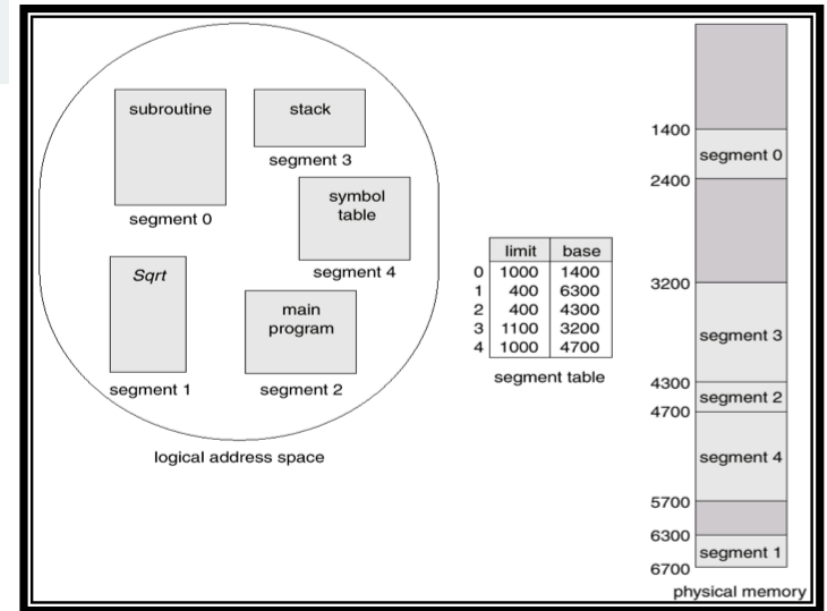


Segmentation pure

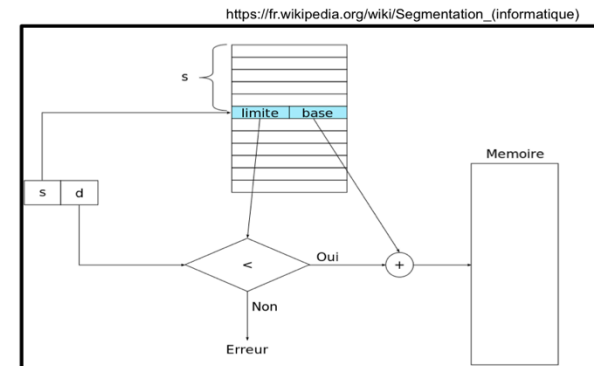
- L'espace d'adressage d'un processus: est un ensemble de **segments de taille différentes, représenté par la table des segments**.
 - Chaque segment est une **zone contiguë**.
 - L'unité d'allocation d'espace mémoire est un segment (zone contiguë).
- Il y a un risque de **fragmentation externe**
- Format de l'adresse virtuelle :

Numéro de segment

Déplacement



Silberschatz



Segmentation pure - Exemple

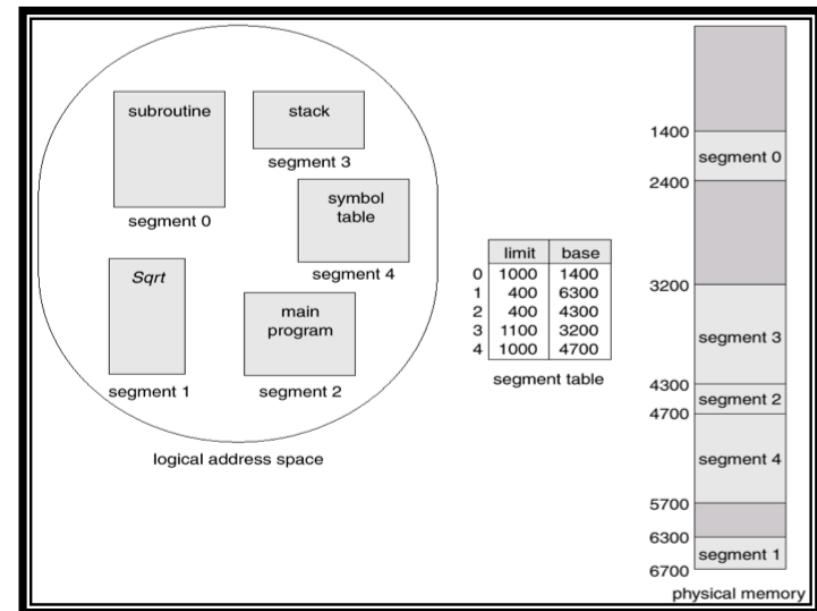
- Soit l'adresse virtuelle

3

200

- Test de validité :
Segment 3 existe et $200 \leq 1100$ OK!
- L'adresse de base du segment 3 est 3200
- L'adresse physique est $3200 + 200 = 3400$

Silberschatz



Segmentation avec pagination

- L'espace d'adressage d'un processus est un **ensemble de segments** et **chaque segment est un ensemble de pages**.
- Il est représenté par une **table de segments** et une **table de pages par segment** (système MULTICS)
- Il y a un risque de **fragmentations interne et externe**.
- Format de l'adresse virtuelle:

Numéro de segment

Numéro de page

Déplacement dans la page



Exercice - Pagination pure

```
./run.sh MemVirt/CompositionAddr
```

Complétez le code suivant afin de pouvoir clairement identifier le numéro de page et le déplacement dans la page d'une adresse virtuelle.

Voir: Capsule Adresses



Exercice - table de pages à plusieurs niveaux

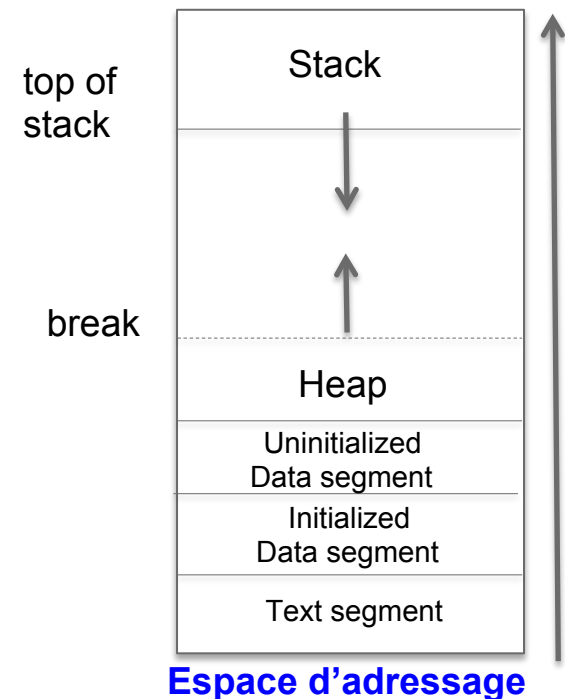
Considérez un système de pagination pure à 2 niveaux dans lequel les adresses virtuelles sont codées sur 32 bits et la taille d'une page est 4KiO. ~~L'entrée de chaque table des pages, peu importe le niveau, est de 8 octets.~~

- Quelle est la taille maximale en nombre de pages de l'espace d'adressage d'un processus ?
- Donnez le nombre de pages qu'il y a dans un espace d'adressage virtuel de 4MiO.
- Donnez le numéro de page qui correspond à l'adresse virtuelle 0x00110A10.

Gestion de l'espace d'adressage d'un processus

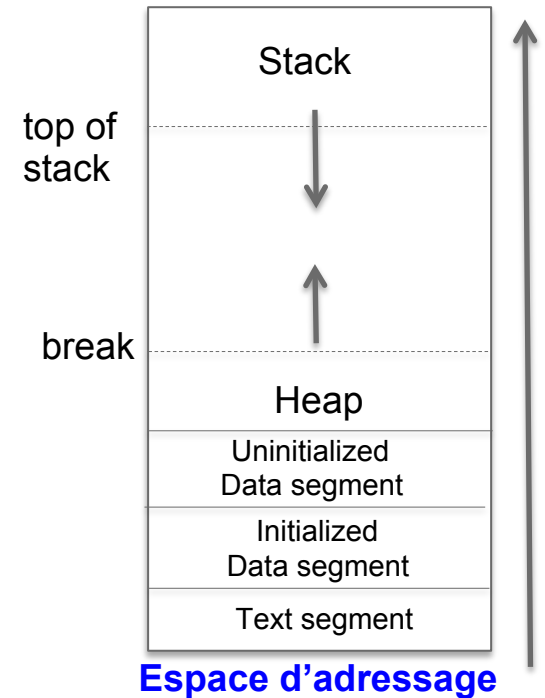
L'espace d'adressage d'un processus est un espace virtuel généré, en partie, par le compilateur et les éditeurs de liens. Cet espace est divisé en segments :

- Text segment: composé des instructions du processus (en lecture seule).
- Initialized data segment: contient les variables/ constantes globales et statiques qui sont initialisées explicitement.
- Uninitialized data segment: contient les variables globales et statiques non initialisées (pages zéro).



Gestion de l'espace d'adressage d'un processus

- Stack (pile) : sert à gérer les appels aux fonctions. Elle est composée de sections allouées et libérées dynamiquement. A chaque appel de fonction une section est empilée dans la pile. Elle est dépilée à la fin de l'appel. Chaque section contient :
 - Un espace réservé à la sauvegarde du contexte de la fonction appelante (l'adresse de retour, le sommet de pile, etc.)
 - les arguments et les variables locales de la fonction.La taille de la pile est limitée mais elle peut être augmentée ***setrlimit*** jusqu'à une certaine limite ***getrlimit***.

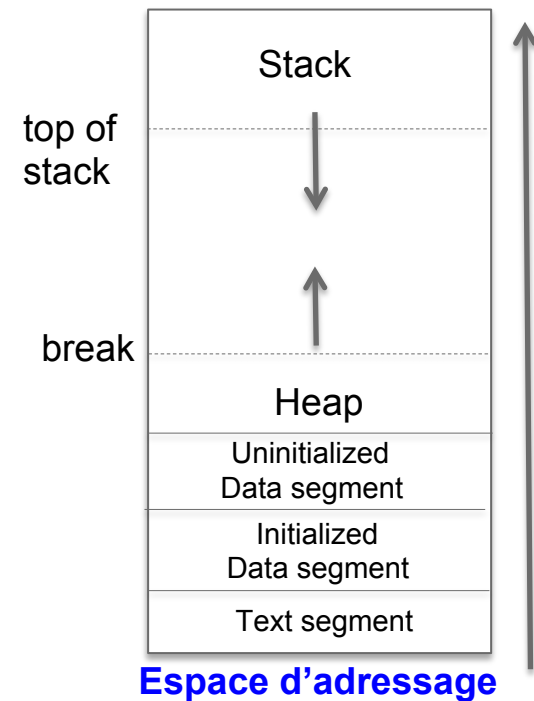


? Gestion de la pile → voir MemVirt/GestionPile

? Épuisement d'espace dans la pile → voir MemVirt/SegfaultHandler

Gestion de l'espace d'adressage d'un processus

- Heap (tas): composé de zones contiguës **allouées/libérées dynamiquement** par les fonctions malloc, realloc, calloc, brk, sbrk, free, etc. Les fonctions brk et sbrk permettent d'incrémenter break (la limite supérieure du heap). **Les allocations explicites doivent être libérées explicitement** (attention aux fuites mémoires).
- Segments entre le tas et la pile contenant les bibliothèques partagées, les piles des threads et les fichiers mappés.

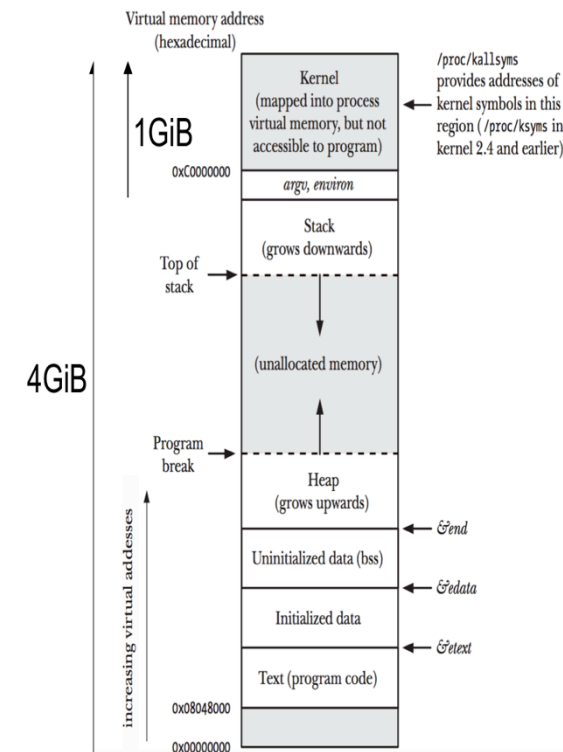


? Gestion du tas → voir MemVirt/SbrkBehaviour

Gestion de l'espace d'adressage d'un processus

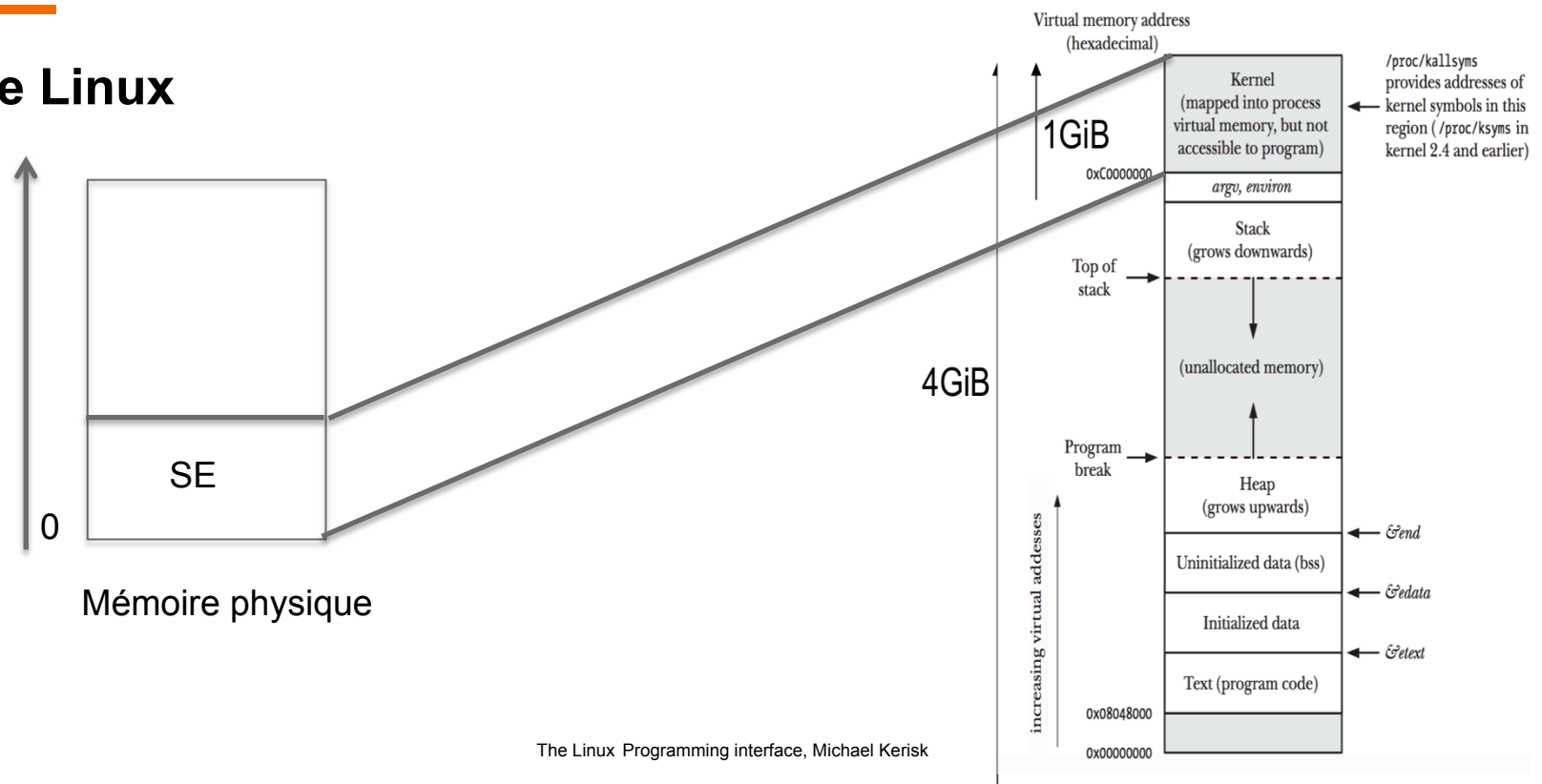
Cas de Linux

- Dans le cas de linux, sur une machine de 32 bits, l'espace d'adressage d'un processus est de 4Gio et est composé de :
 - 3 Gio pour le code, les données, et les piles d'exécution du processus, accessible en mode utilisateur et noyau
 - 1 Gio pour les tables des pages, la pile d'exécution, les données et le code du SE, accessible en mode noyau uniquement.



Gestion de l'espace d'adressage d'un processus

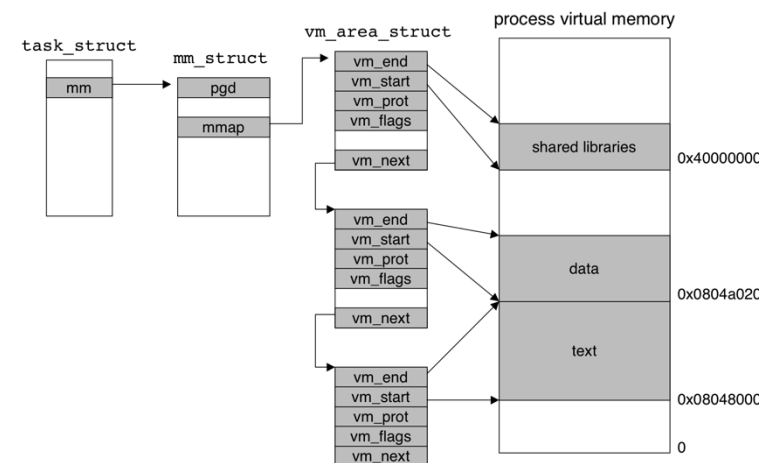
Cas de Linux



Gestion de l'espace d'adressage d'un processus

Cas de Linux

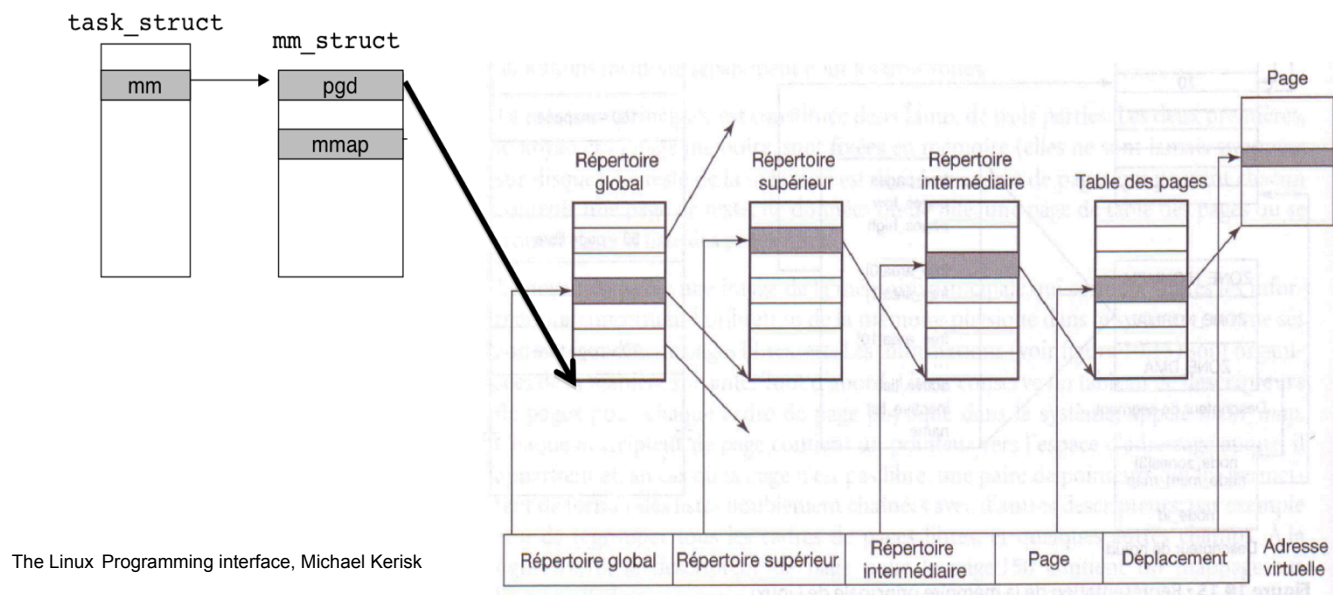
- L'espace d'adressage est représenté par :
 - une liste (ou un arbre binaire de recherche) de structures `vm_area_struct` (une structure par région) et
 - une table des pages à plusieurs niveaux, etc.
- La structure `vm_area_struct` est composée de :
 - `vm_start` qui pointe vers le début de la région,
 - `vm_end` qui pointe vers la fin de la région,
 - `vm_prot` qui indique les permissions d'accès r/w/x,
 - `vm_flags` qui indique, notamment si la région est **privée ou non**, etc.



The Linux Programming interface, Michael Kerisk

Gestion de l'espace d'adressage d'un processus

Cas de Linux



Gestion de l'espace d'adressage d'un processus

Cas de LINUX - Quelques appels système

- **fork()** crée un processus fils (duplication de l'espace d'adressage du père selon le principe Copy-On-Write).
- **execl()** remplace l'espace d'adressage du processus appelant par un autre, construit à partir de l'exécutable spécifié par le premier paramètre de execl.
- **exit()** libère l'espace d'adressage d'un processus.
- **mmap()** crée un segment dans l'espace d'adressage du processus appelant
- **munmap()** supprime un segment de l'espace d'adressage d'un processus (entièrement ou en partie).

Gestion de l'espace d'adressage d'un processus

Cas de LINUX - Exemple : appel système fork

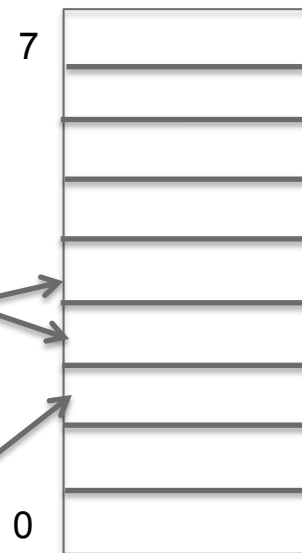
- fork crée, pour le processus fils, une copie de mm_struct, vm_area_struct et des tables de pages du père en positionnant les bits de protection de chaque page dans les deux processus **à lecture seule**.
- Le principe de Copy-On-Write (COW) est appliqué à **chaque segment (vm_area_struct) privé avec les permissions R/W**.
- Si le père ou le fils demande un accès en écriture à une page chargée en mémoire physique alors ses bits de protection indique qu'elle est en lecture seule, il se produit un **défaut de protection qui a pour but de créer** une copie de cette page en mémoire physique pour le processus demandeur d'accès.

Gestion de l'espace d'adressage d'un processus

Cas de LINUX - Exemple : appel système fork

Avant fork

Table des pages			
Index	Bit de présence	Bits de protection	Numéro de cadre
0	1	r	3
1	1	r/w	4
2	0	r	0
3	0	r/w	0
4	1	r/w	2



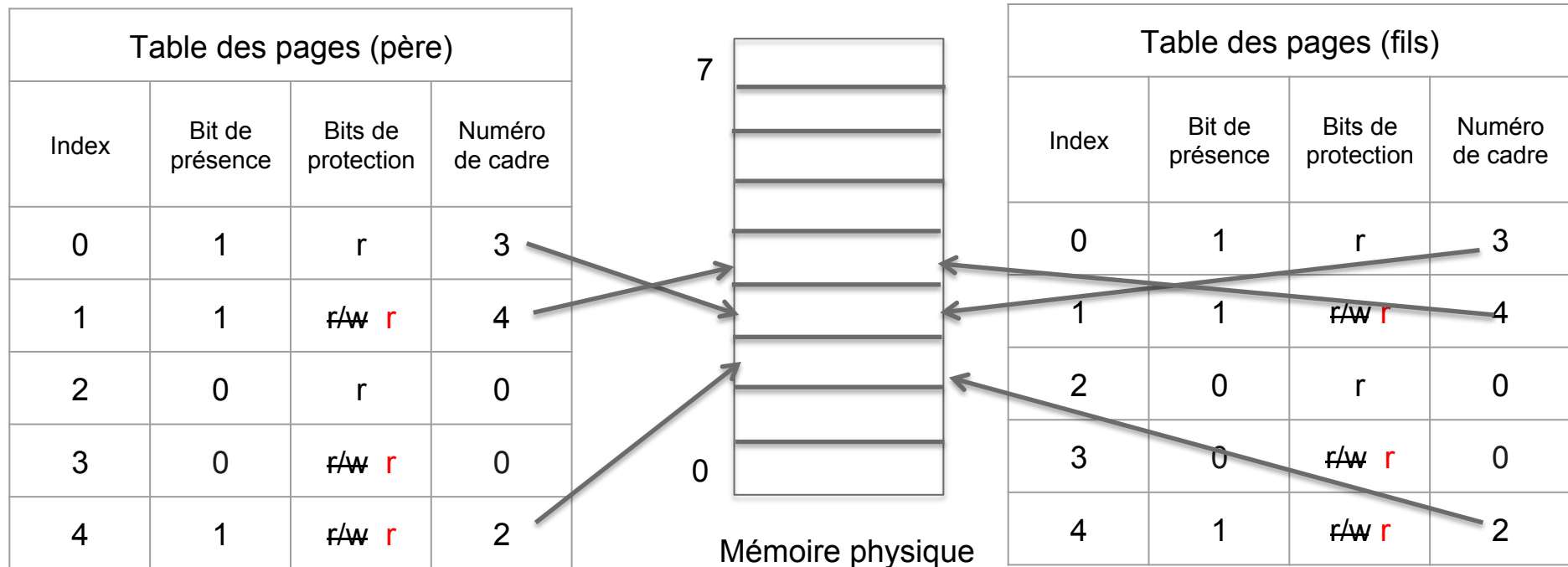
Mémoire physique

On suppose que toutes ces pages sont des régions privées du processus.

Gestion de l'espace d'adressage d'un processus

Cas de LINUX - Exemple : appel système fork

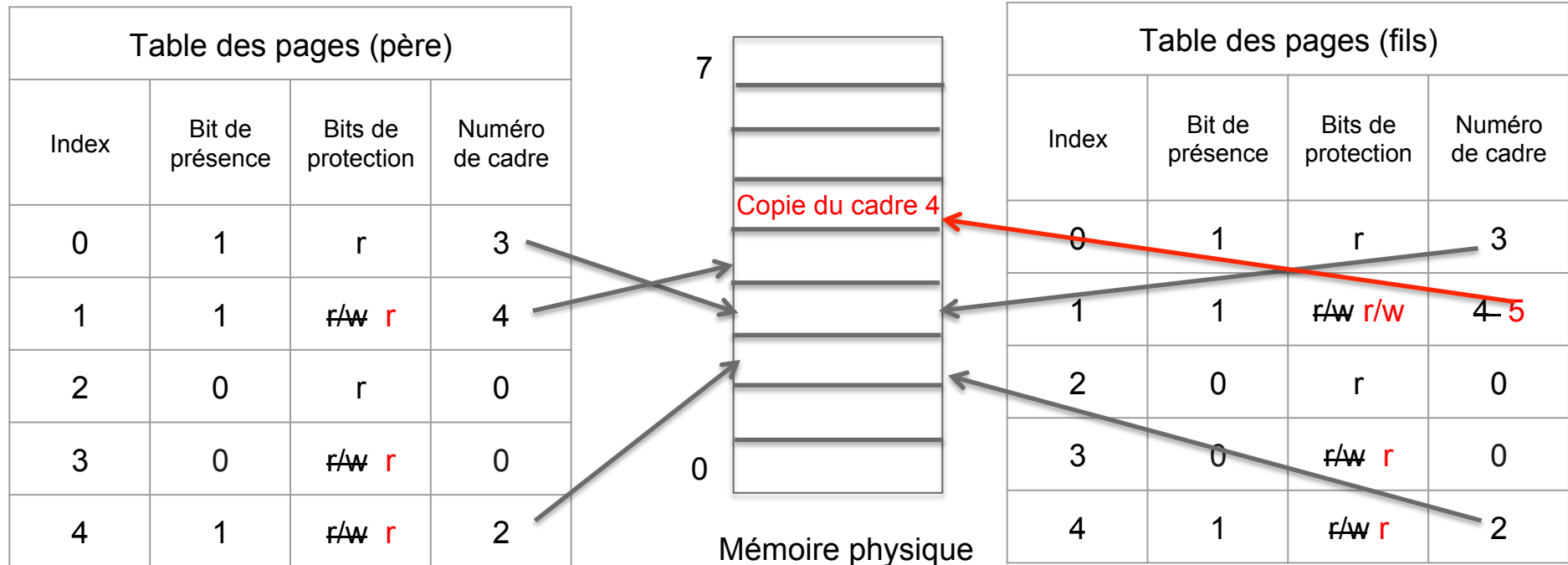
Après fork



Gestion de l'espace d'adressage d'un processus

Cas de LINUX - Exemple : appel système fork

Fils référence la page 1 en écriture (qui est dans un segment privé en RW)





Exemples : Segments de l'espace d'adressage d'un processus

- Testez les codes C dans les répertoires : codes/MemVirt/AddressesSections et codes/MemVirt/Addresses.
- Complétez puis testez le code C dans le répertoire : /MemVirt /CompositionAddr.
- Comment récupérer les segments (régions) d'un espace d'adressage ?
- Comment déterminer dans quelle région se trouve une fonction, une donnée ou une adresse ?
- Comment extraire le numéro de page et le déplacement à partir d'une adresse et d'une taille de page (en décimal) ?



Exemple Gestion de la pile

Consultez les codes dans le répertoire :codes/MemVirt/GestionPile

- Cet exemple est utilisé pour vous montrer comment la pile d'exécution est gérée.
- L'évolution de la pile pour cet exemple est montré, pas à pas, en déroulant le code assembleur gestion-pile.s (obtenu par la commande **gcc -S gestion-pile.c**).
- Pour gérer les appels aux fonctions deux registres sont utilisés :
 - rbp qui pointe sur la base de la section courante et
 - rsp qui pointe sur le sommet de la section courante (sommet de pile).

```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```

```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```

```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```

```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```

```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```

```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```

```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```

```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```

```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```

```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```

```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```

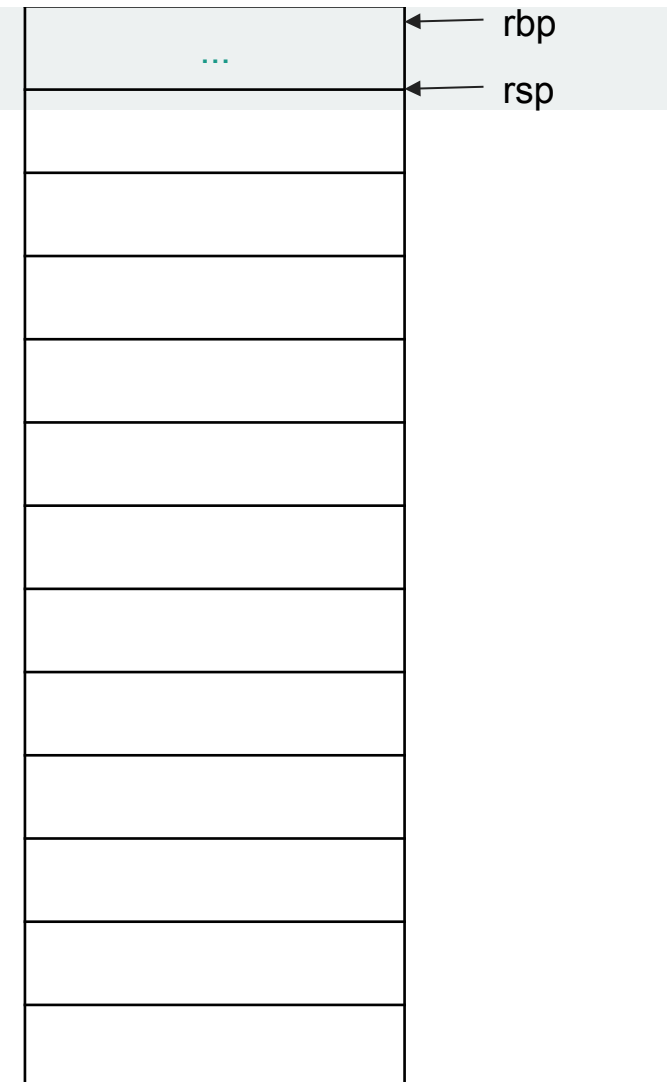
```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```

```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```

```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```

```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```

```
main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret
```



```
main:
```

```
.LFB1:
```

```
pushq    %rbp
```

```
movq     %rsp, %rbp
```

```
subq     $16, %rsp
```

```
movl     $2, -4(%rbp)
```

```
movl     -4(%rbp), %eax
```

```
movl     %eax, %edi
```

```
call     f
```

```
movl     %eax, %esi
```

```
leaq     .LC0(%rip), %rdi
```

```
movl     $0, %eax
```

```
call     printf@PLT
```

```
movl     $0, %eax
```

```
leave
```

```
ret
```

...

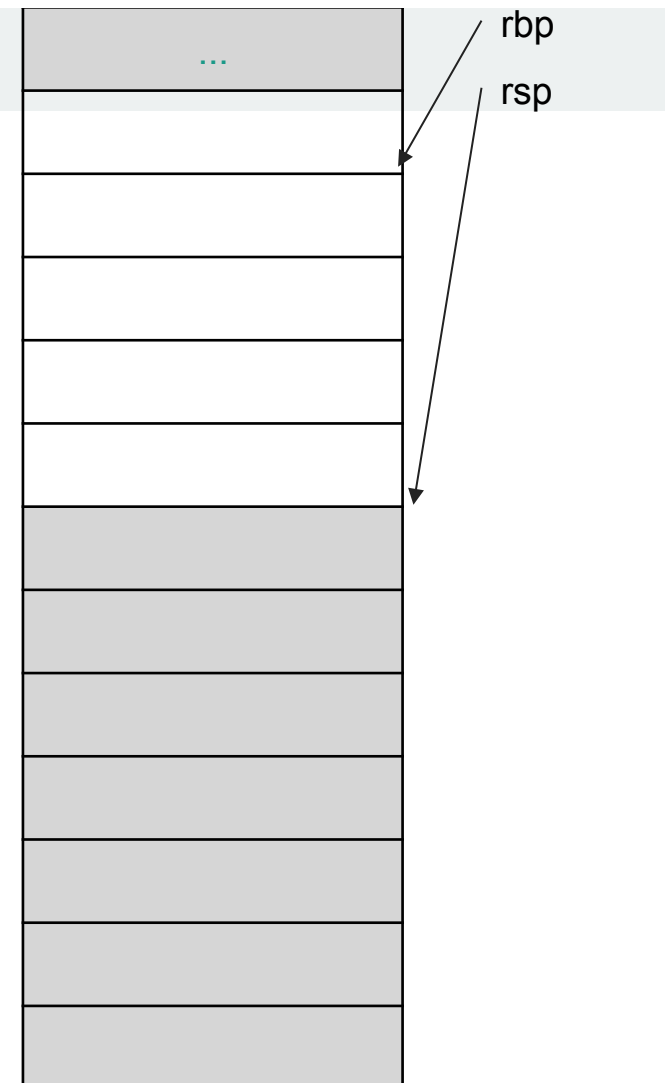
rbp

rsp

```

main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret

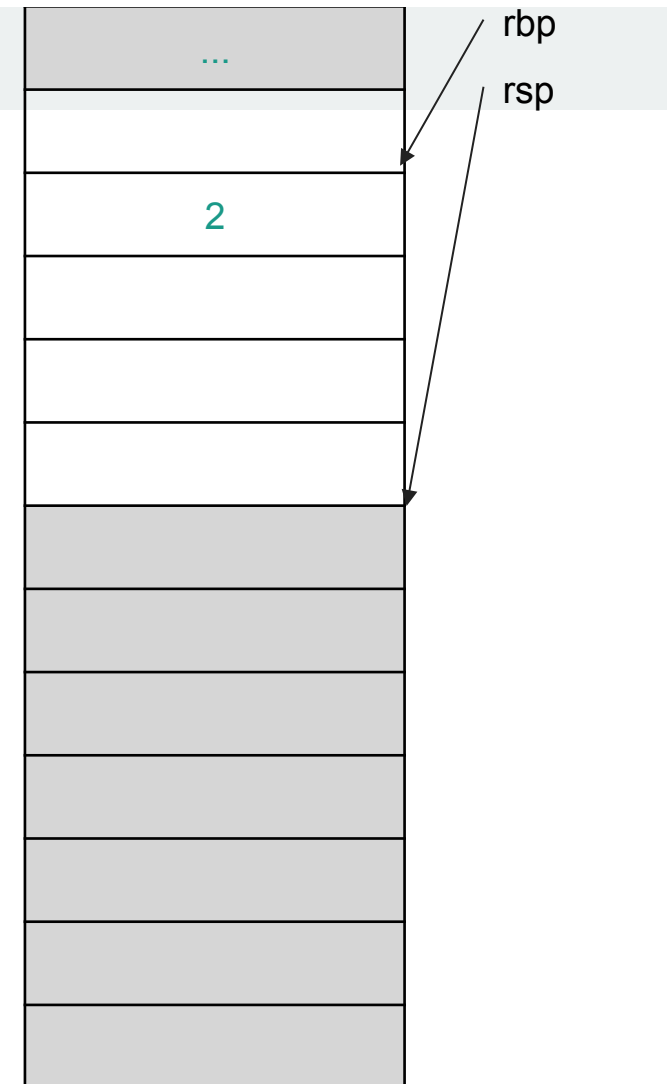
```



```

main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret

```

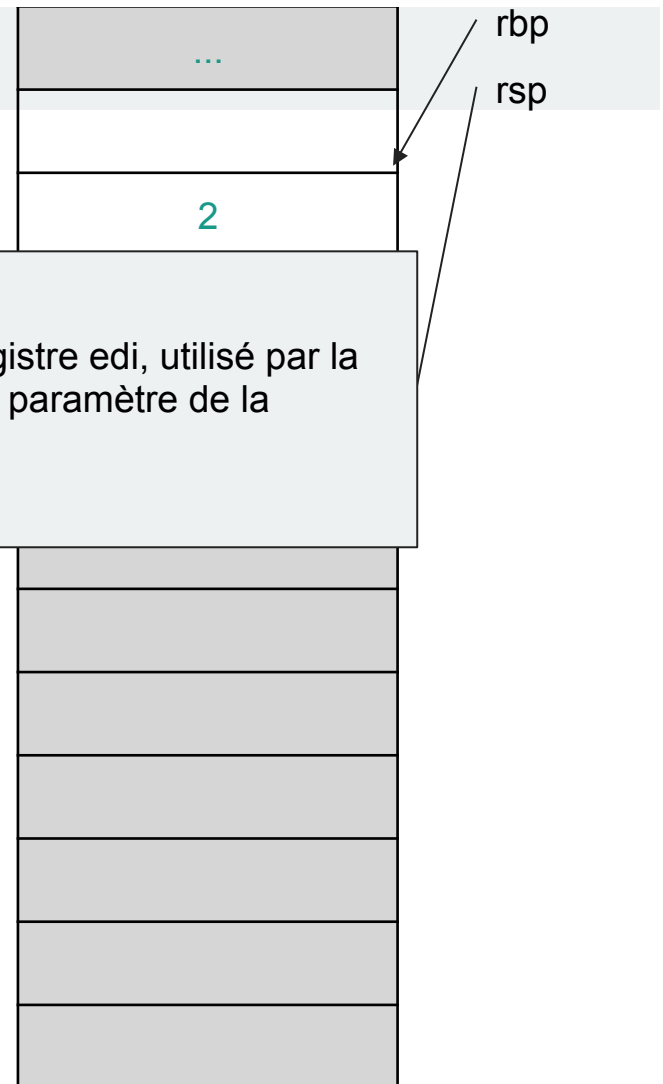


```

main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret

```

On déplace le 2 dans le registre edi, utilisé par la fonction f pour récupérer le paramètre de la fonction.



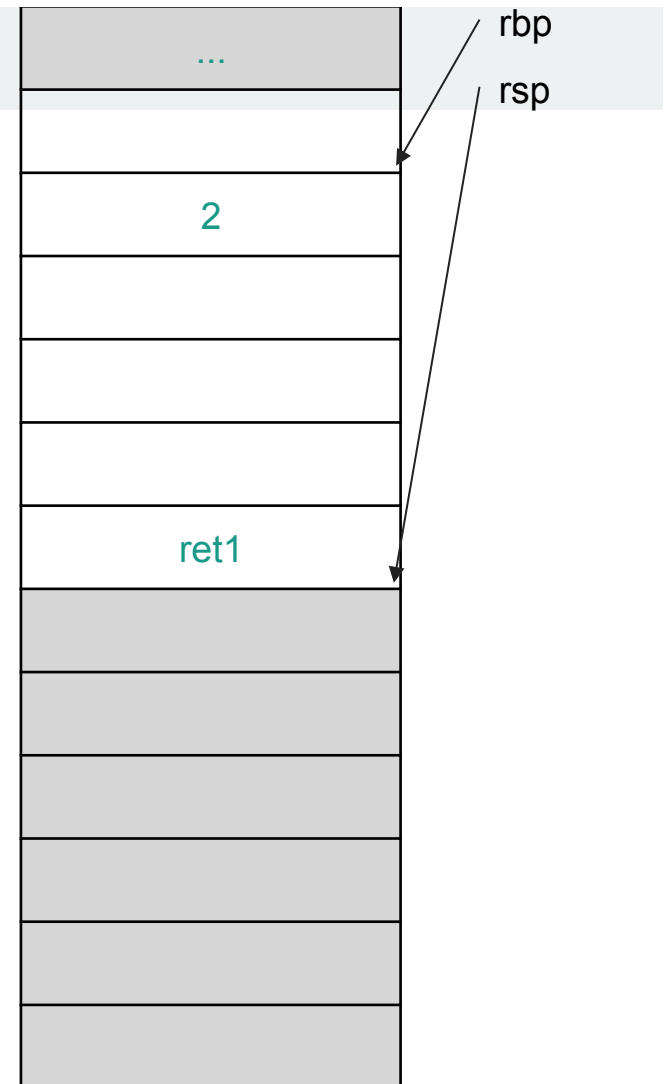
```

main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret

```

push CO
CO ← f

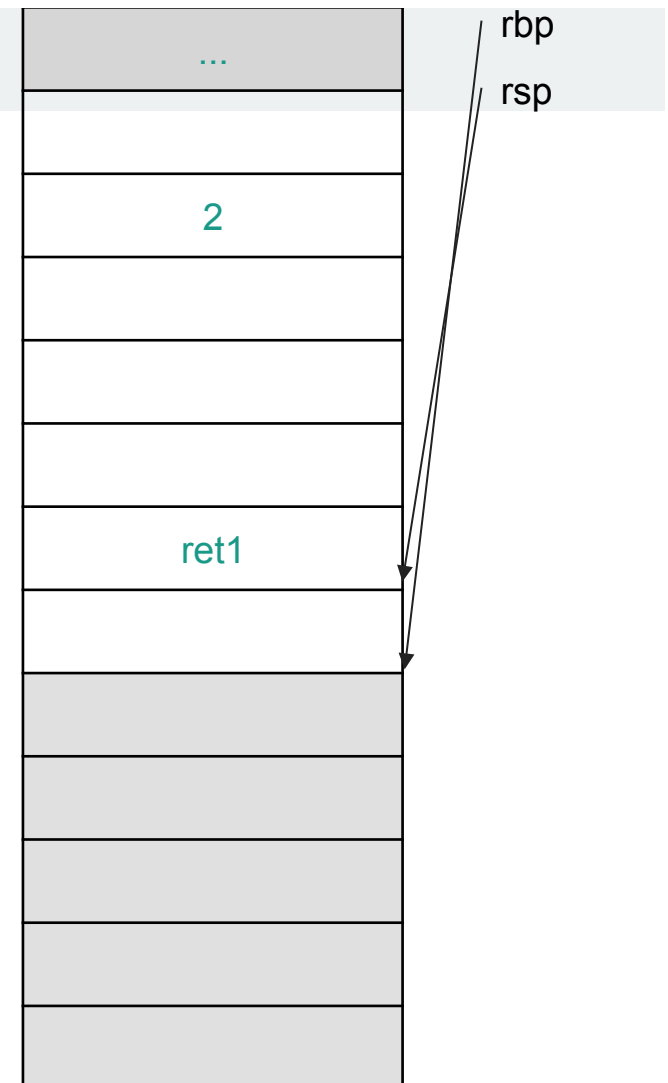
ret1



```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

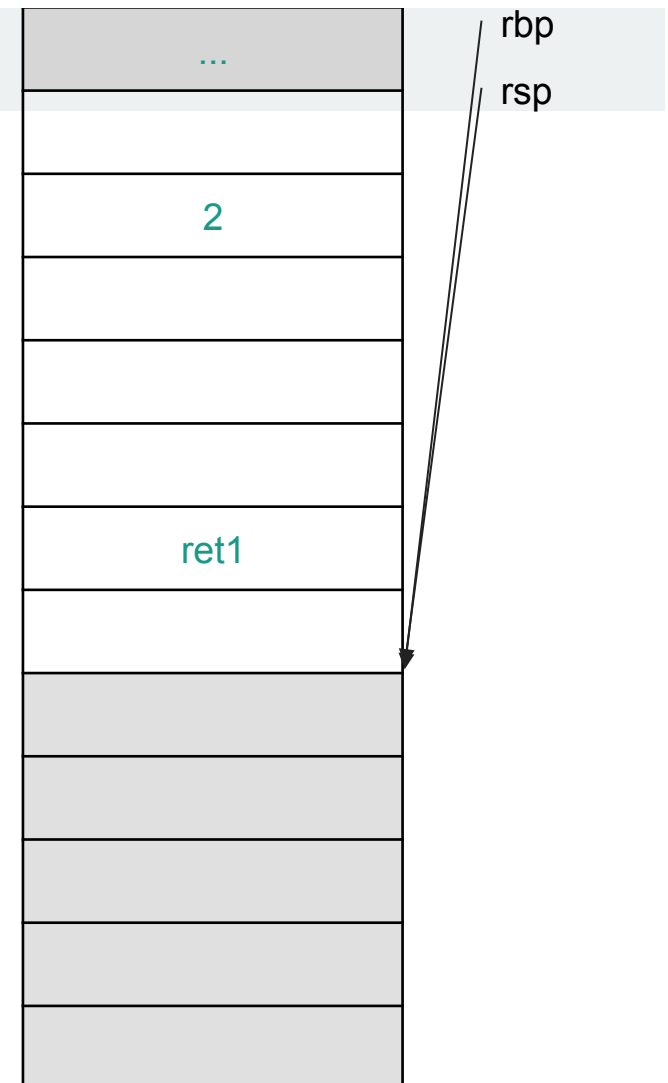
```




```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

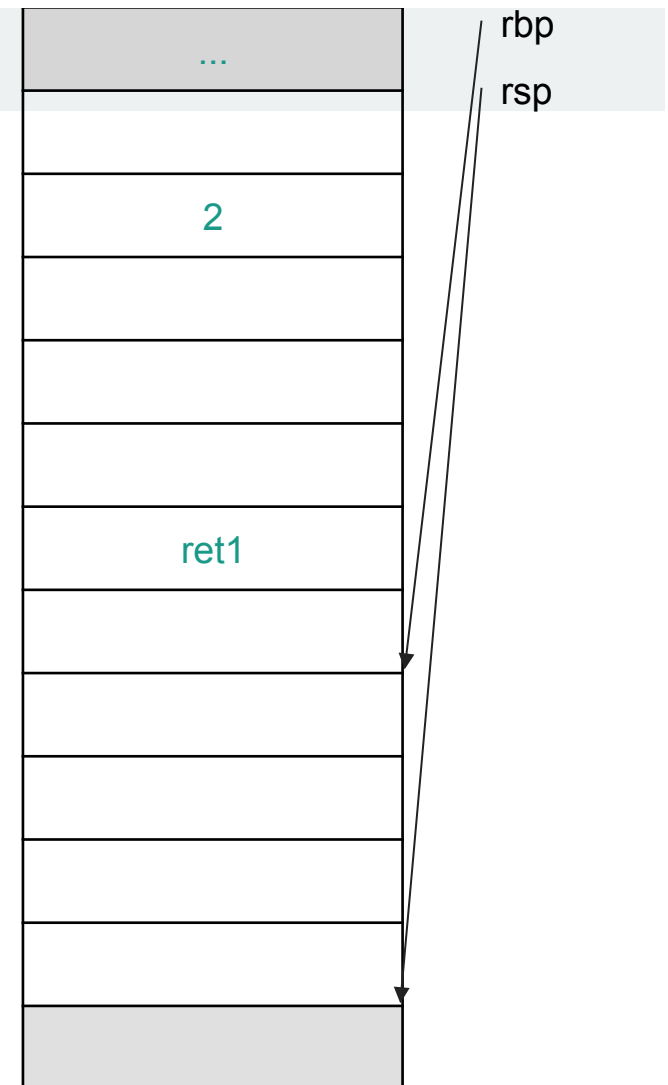
```



```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

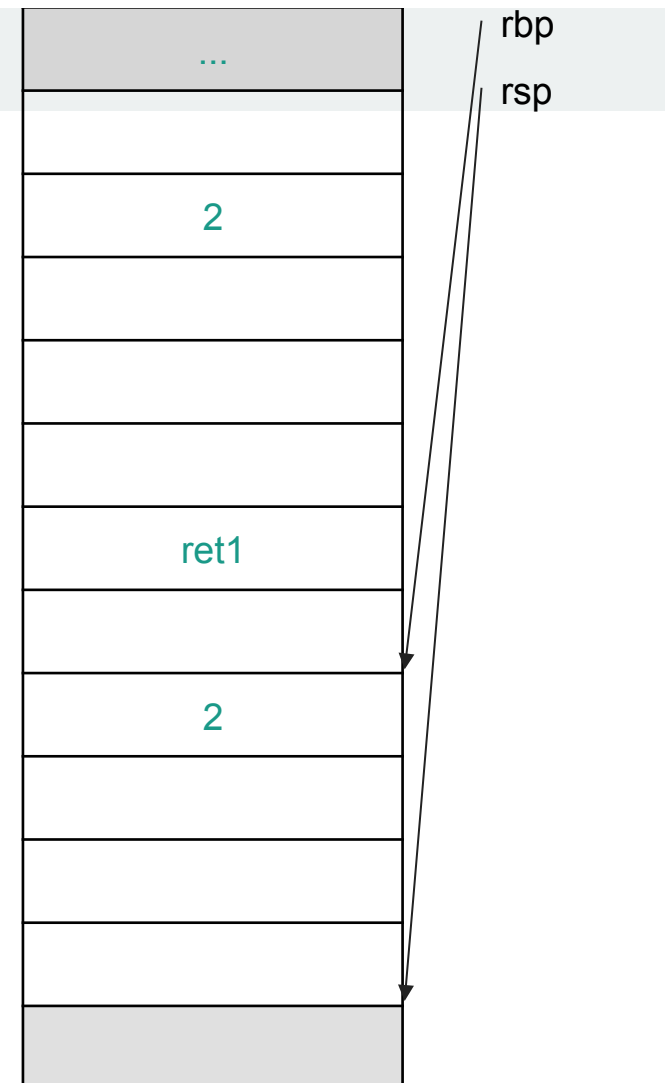
```



```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

```

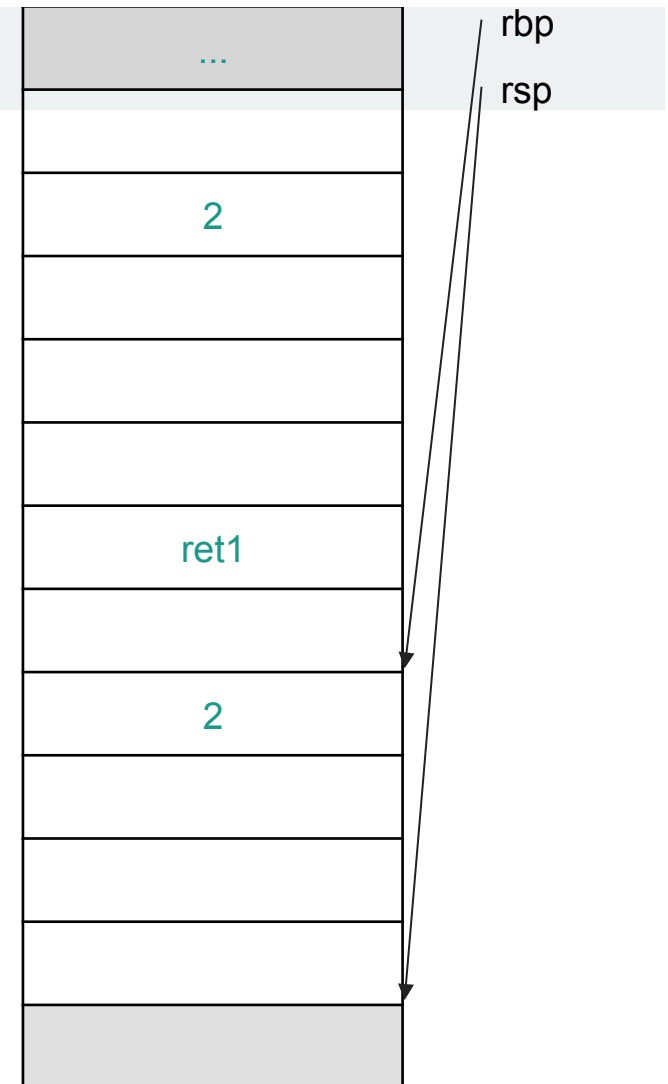


```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

```

On regarde si le paramètre est égal ou non à 1

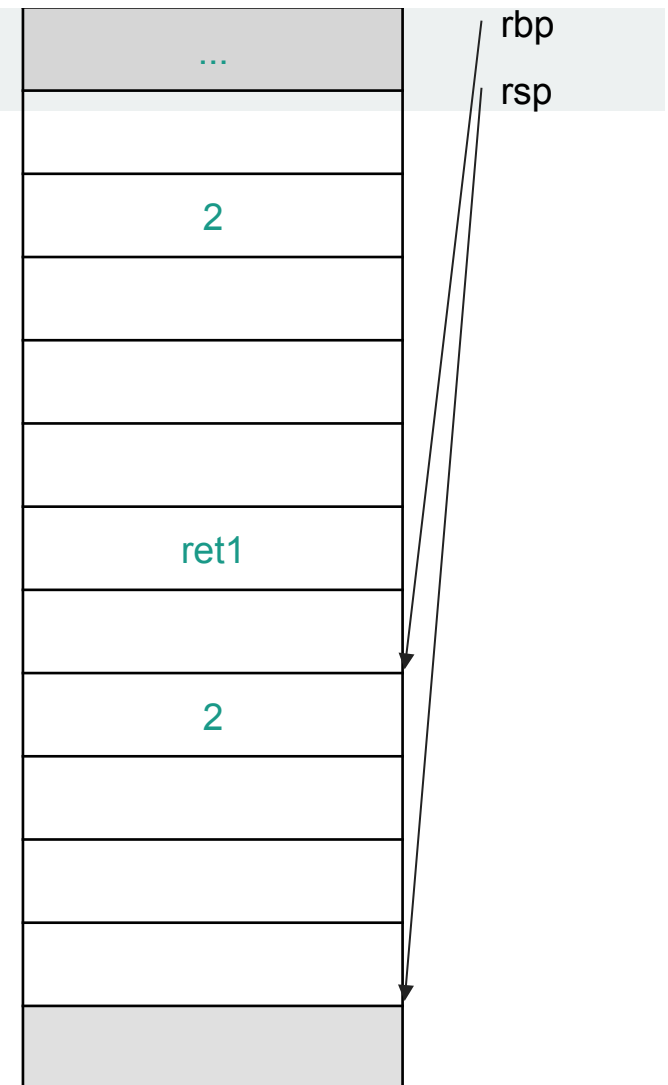


```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

```

Comme la valeur du paramètre est 2, on saute à .L2

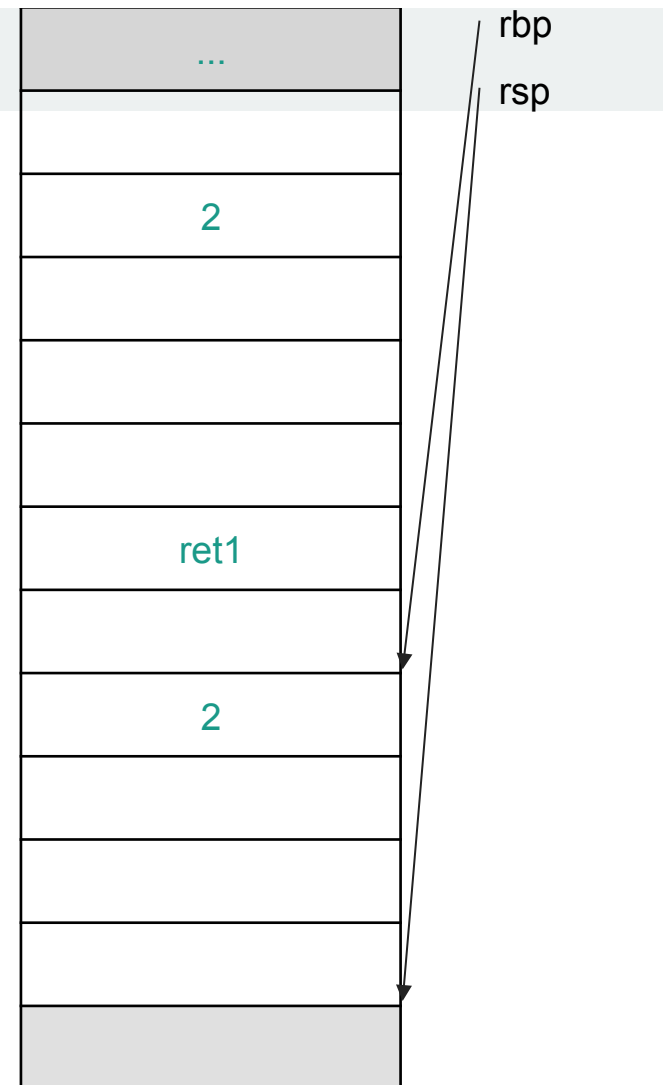


```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

```

On place 2 dans le registre eax

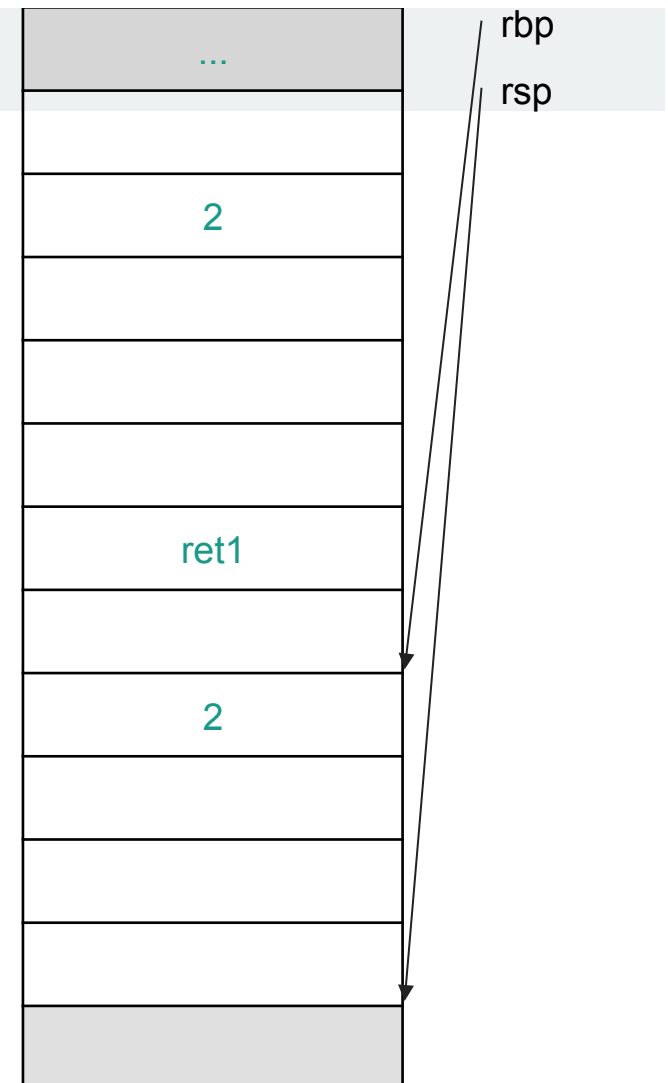


```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

```

On soustrait 1 à la valeur de eax (pour se préparer à appeler f(x-1))

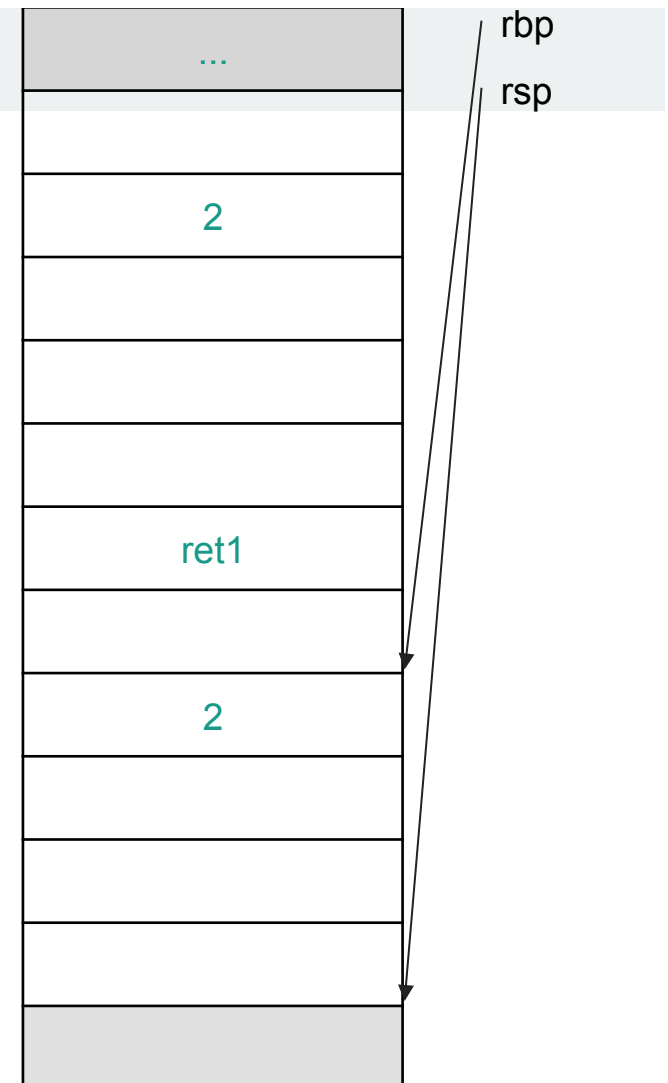


```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

```

On place l'argument
dans le registre edi
utilisé par la fonction f
pour récupérer
l'argument



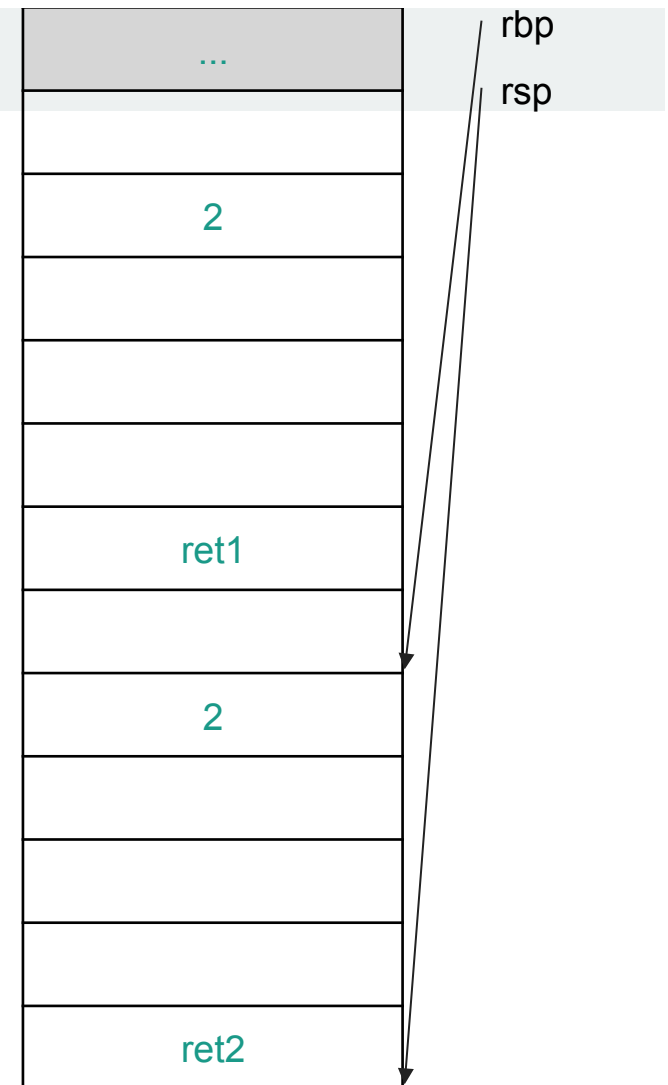

```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

```

On appelle la fonction f
push CO
CO ← f

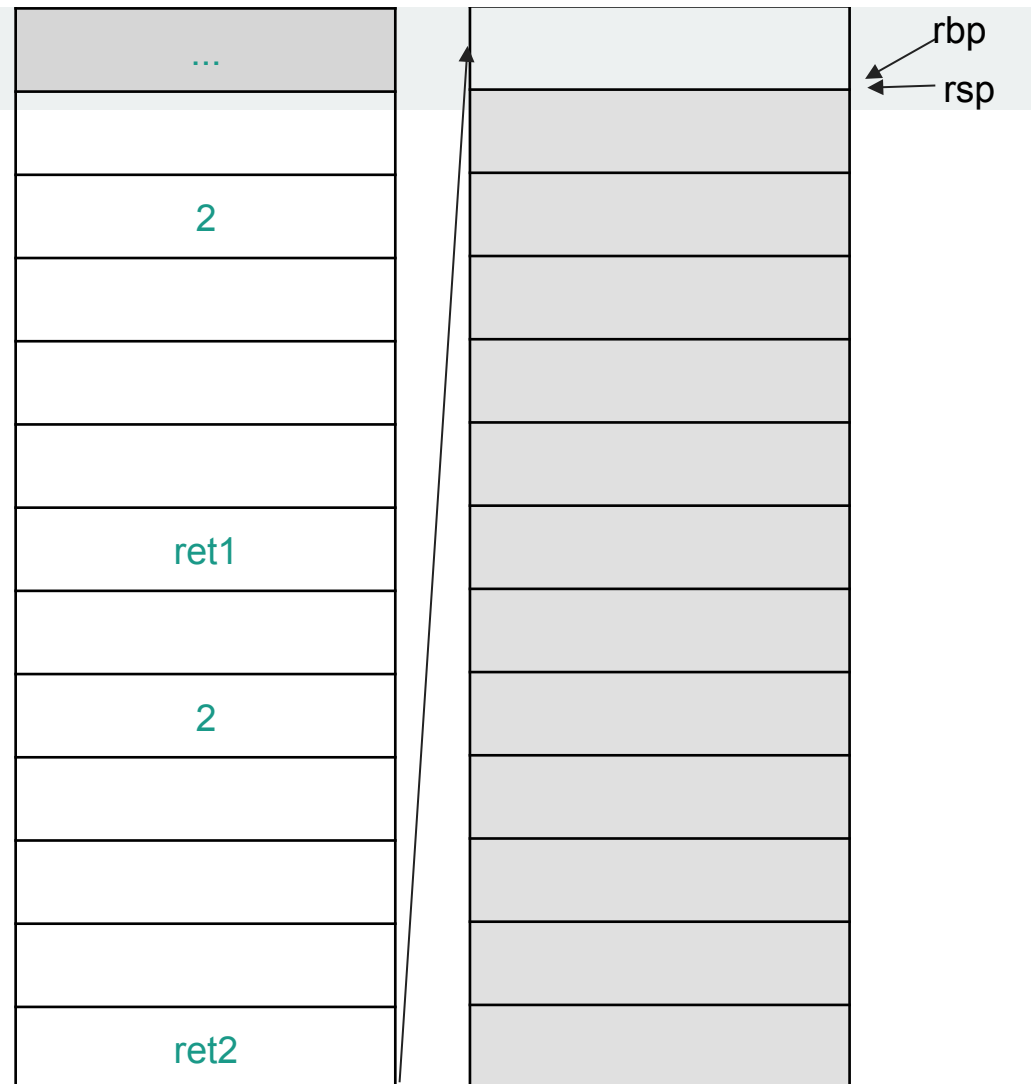
ret2



```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

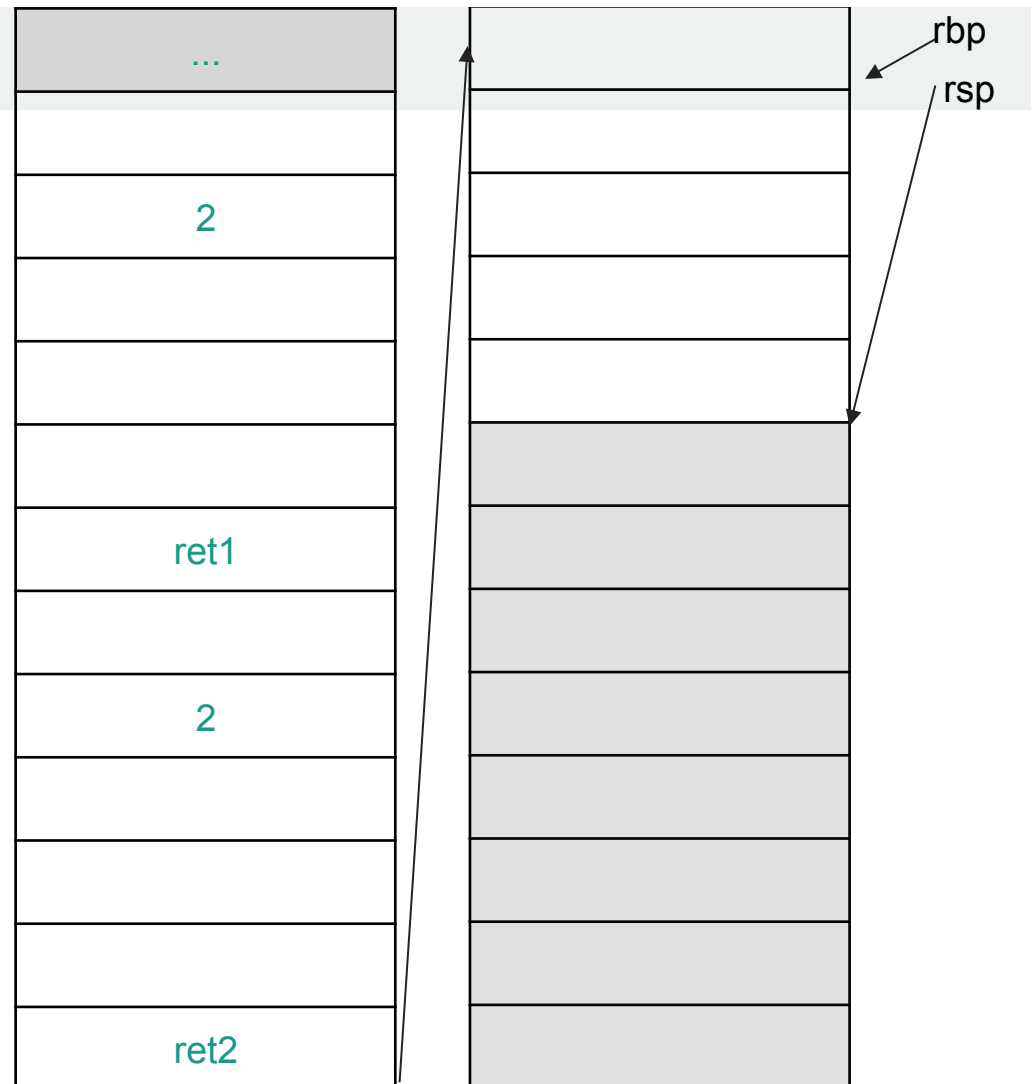
```



```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

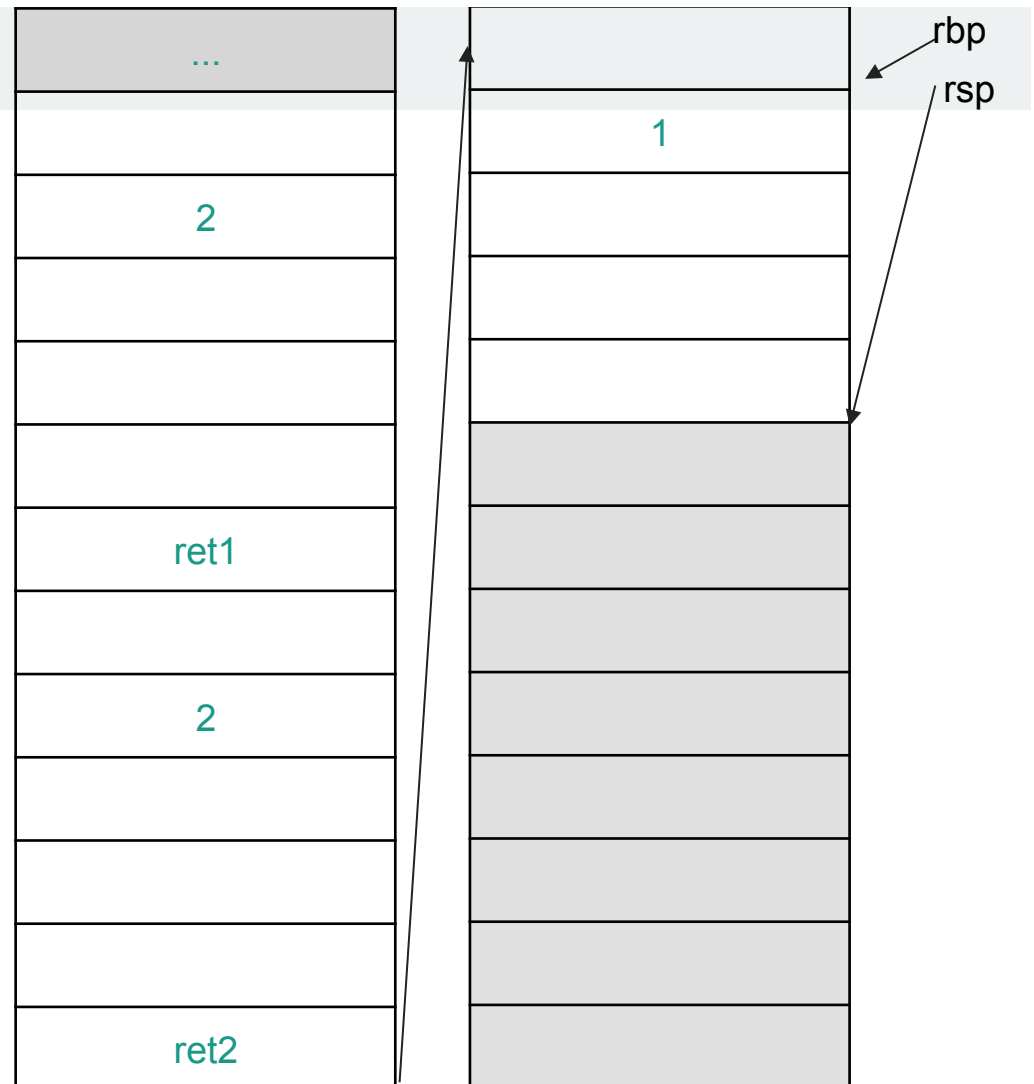
```



```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

```

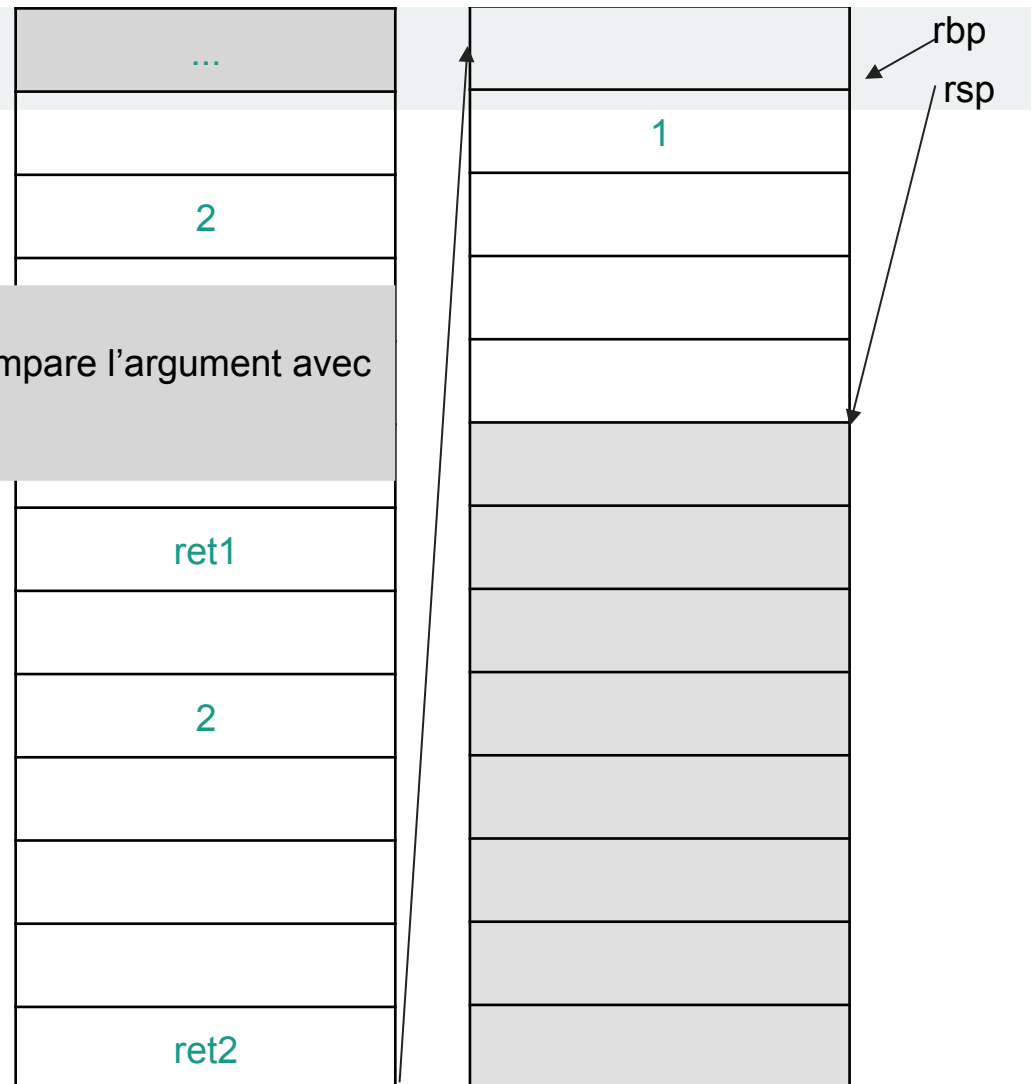


```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

```

On compare l'argument avec 1



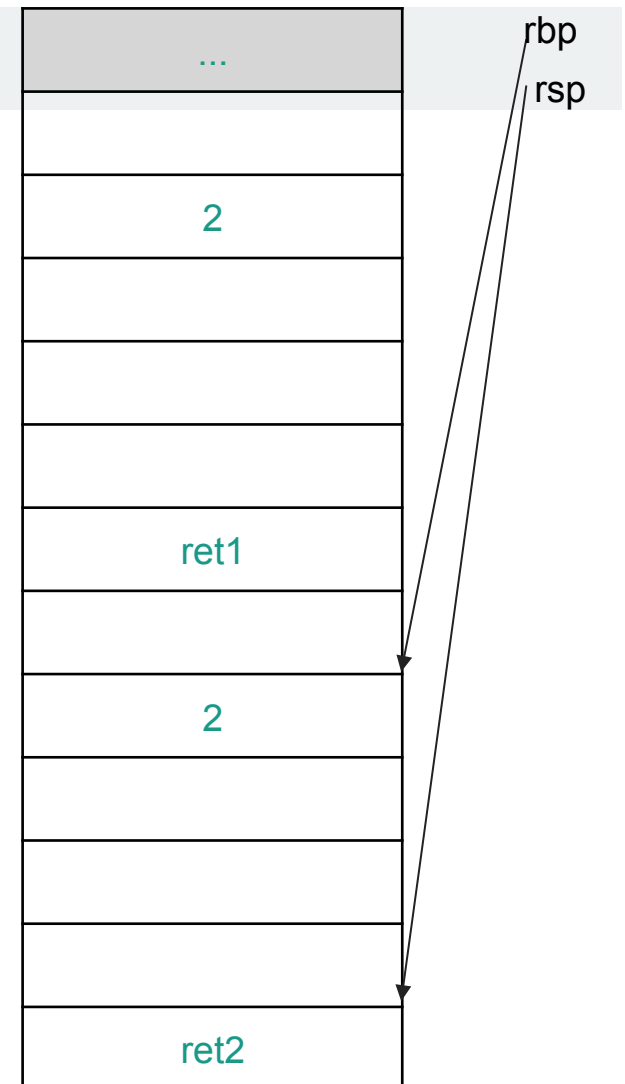
```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

```

L'argument est égal à 1! On place alors 1 dans le registre de retour (eax) et on retourne à l'appelant (ret2)

rbp ← pop()
CO ← pop()

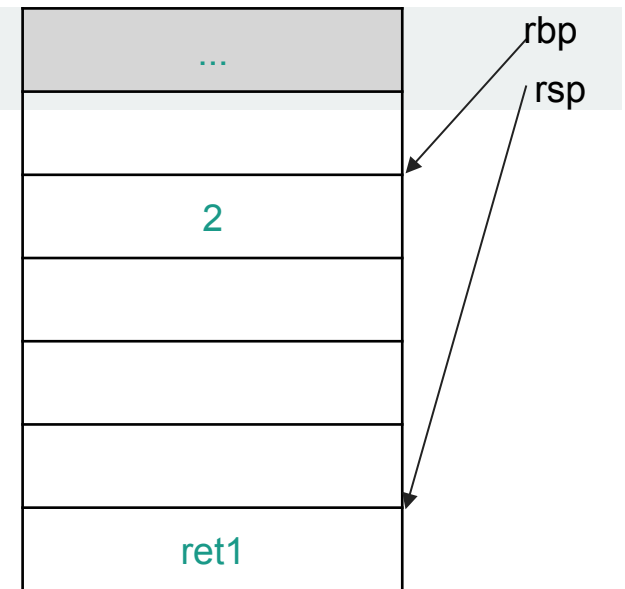


```

f:
.LFB0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     %edi, -4(%rbp)
    cmpl     $1, -4(%rbp)
    jne .L2
    movl     $1, %eax
    jmp .L3
.L2:
    movl     -4(%rbp), %eax
    subl     $1, %eax
    movl     %eax, %edi
    call     f
    imull    -4(%rbp), %eax
.L3:
    leave
    ret

```

On multiplie 2 avec le contenu de eax (1) donc $2 * 1 = 2$ et on place le résultat dans eax avant de retourner à l'appelant (ret1)

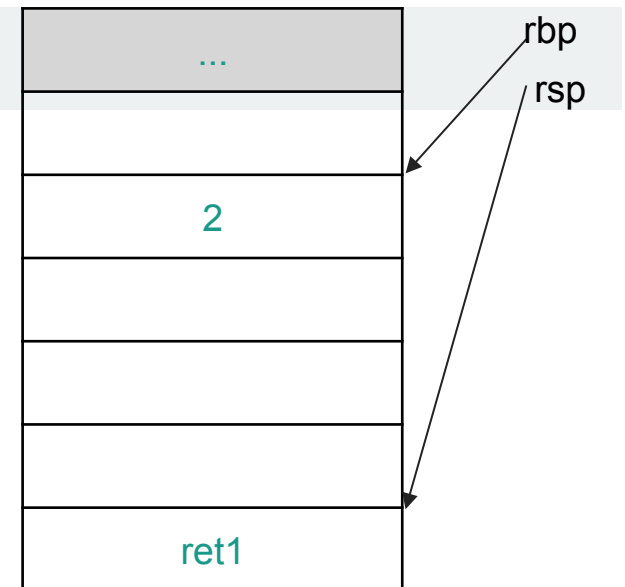


```

main:
.LFB1:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $2, -4(%rbp)
    movl     -4(%rbp), %eax
    movl     %eax, %edi
    call     f
    movl     %eax, %esi
    leaq     .LC0(%rip), %rdi
    movl     $0, %eax
    call     printf@PLT
    movl     $0, %eax
    leave
    ret

```

On fait ensuite les opérations nécessaires pour appeler printf...



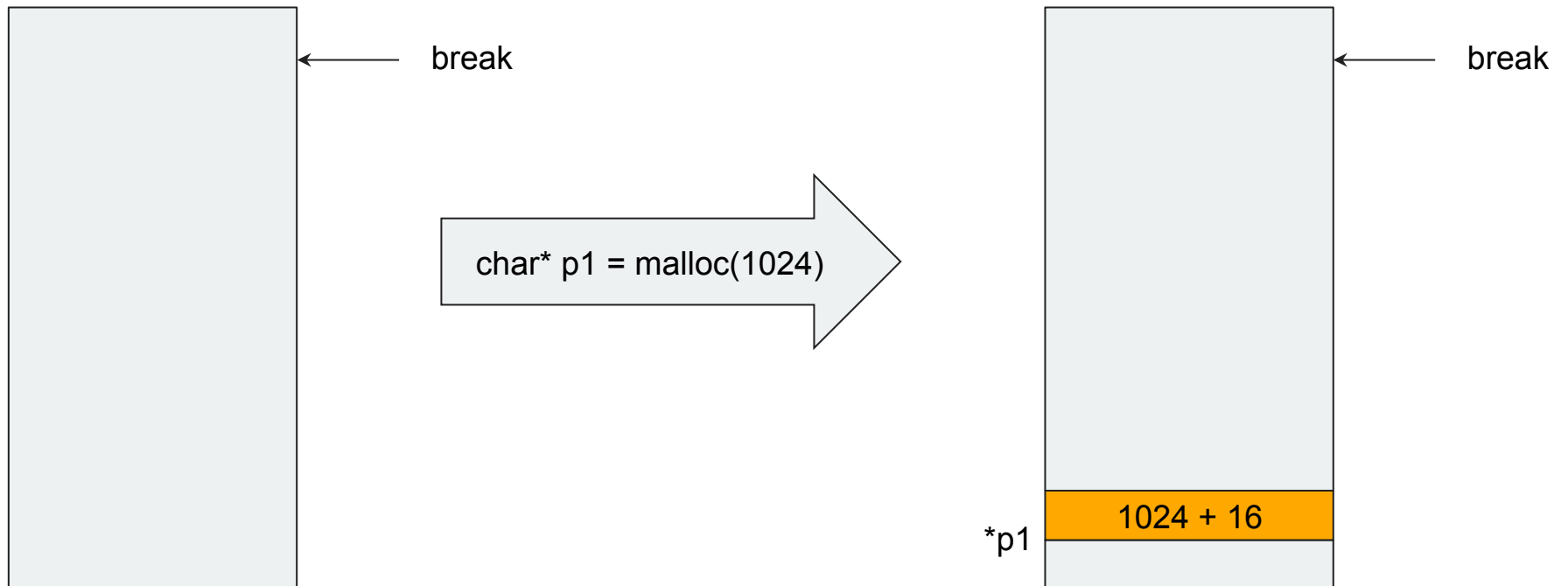


Exemple Gestion du tas (heap)

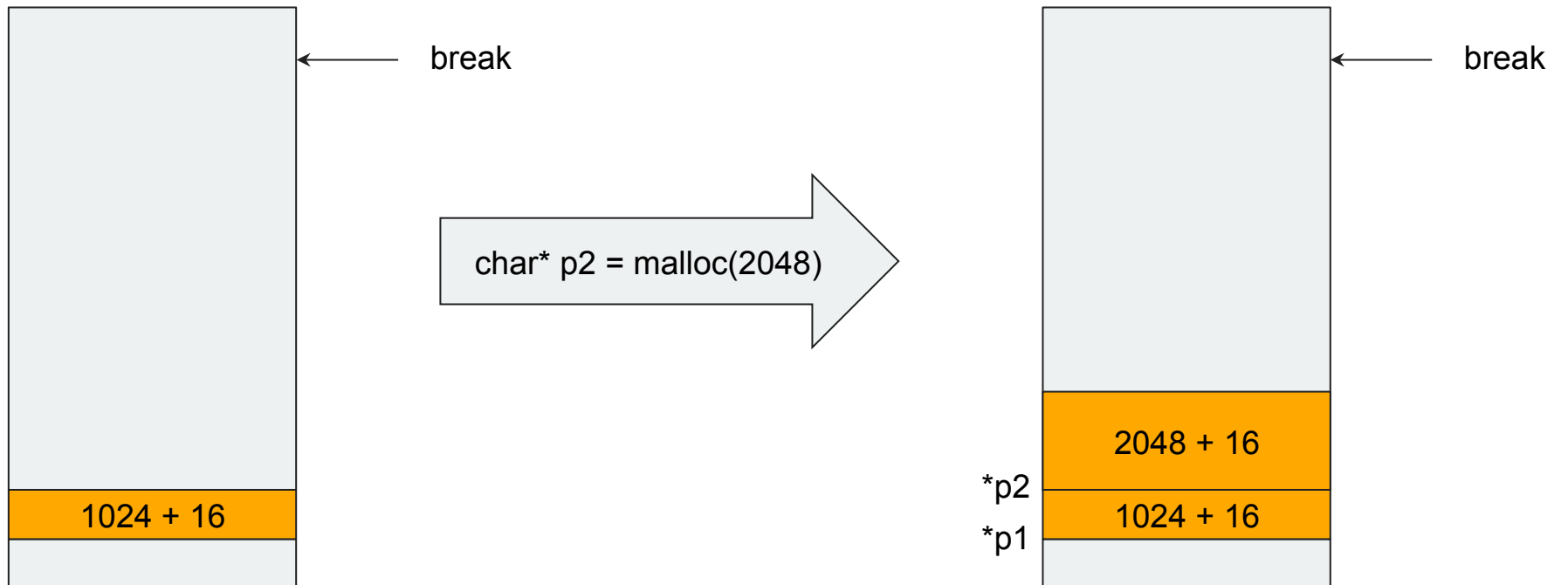
Testez le code dans le répertoire :
`codes/MemVirt/SbrkBehaviour`

- Comment C gère-t-il le tas sous Linux ?

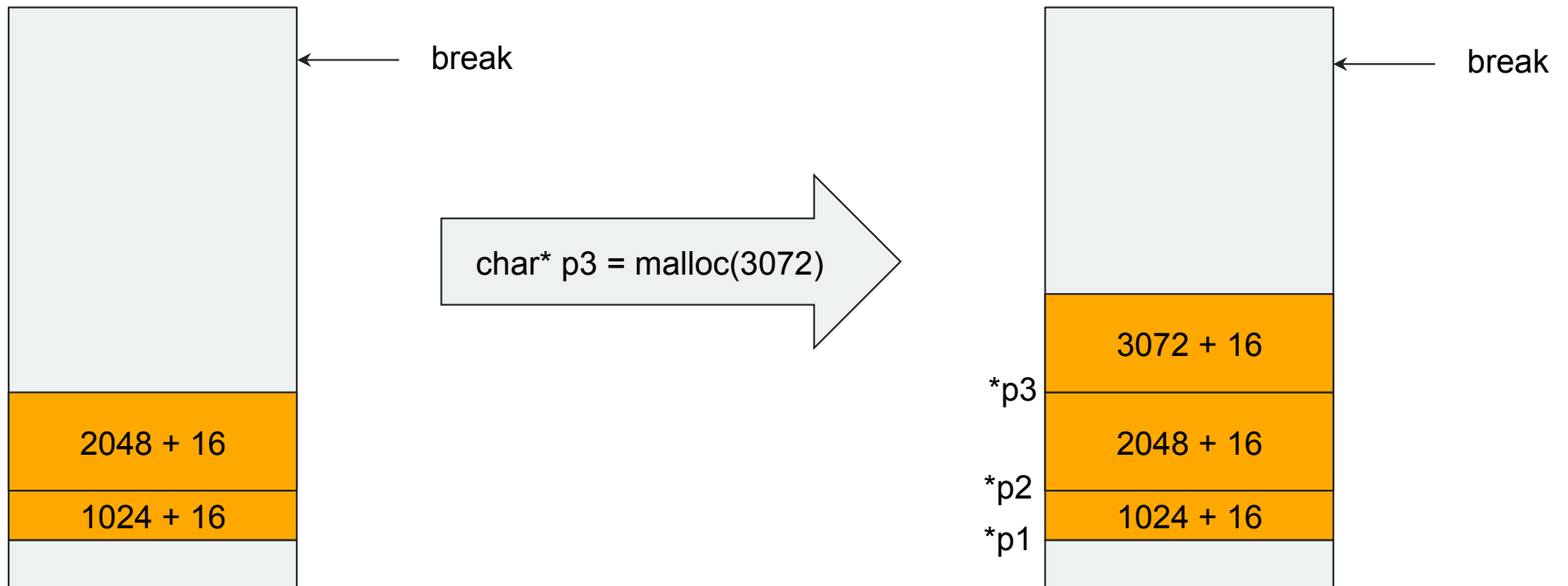
Gestion du tas (heap)



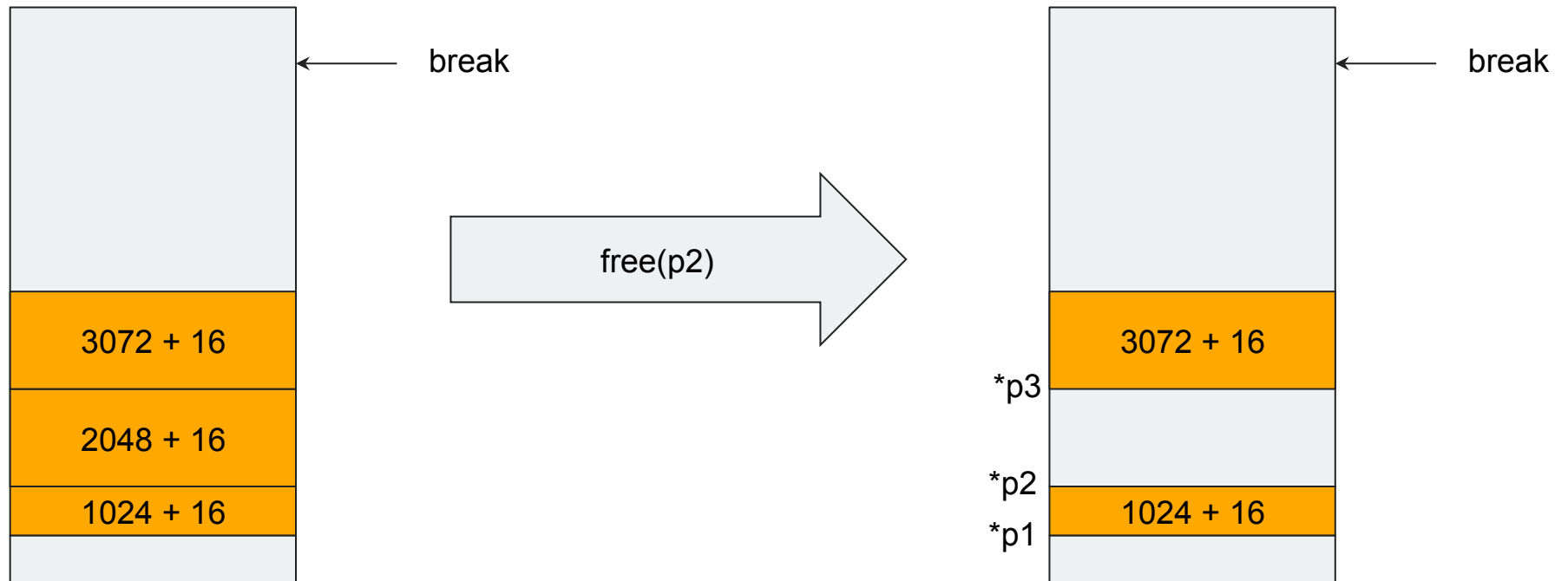
Gestion du tas (heap)



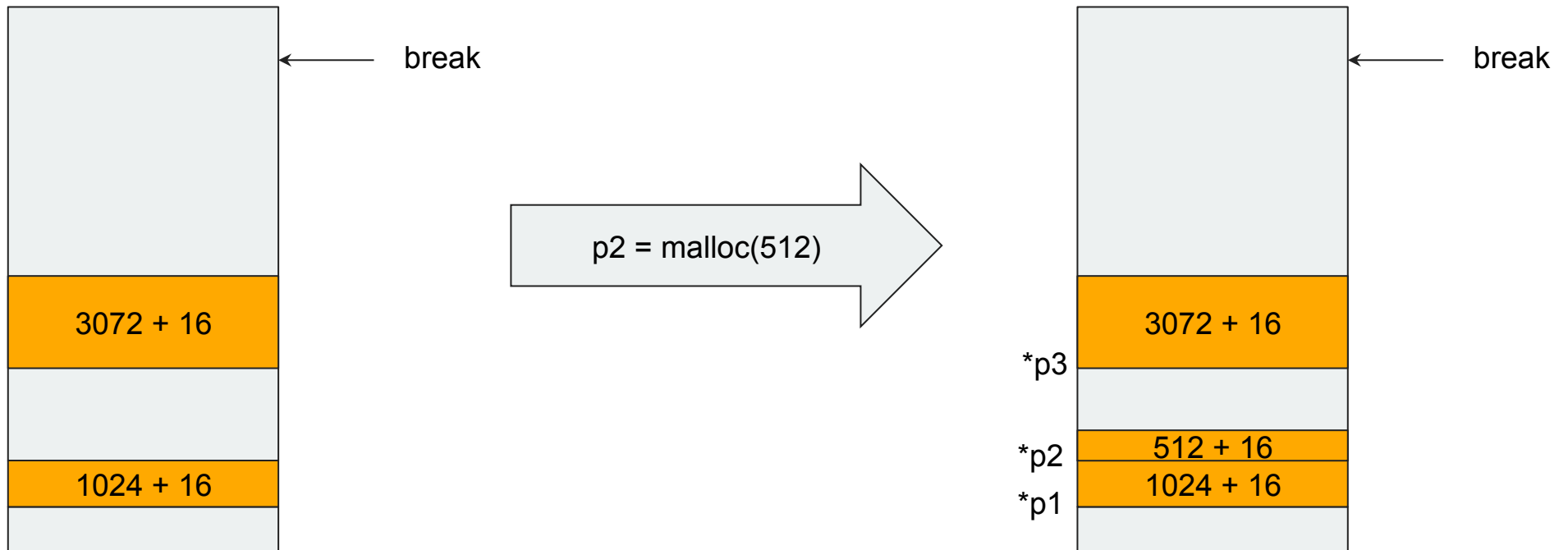
Gestion du tas (heap)



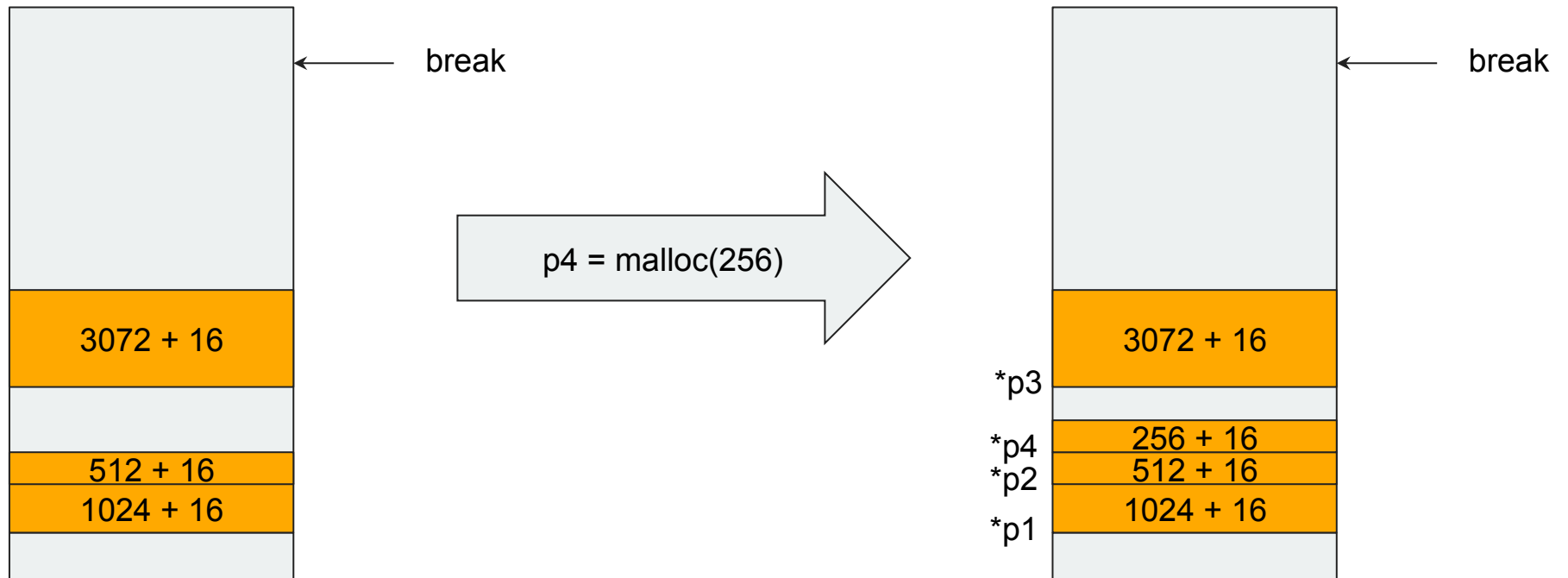
Gestion du tas (heap)



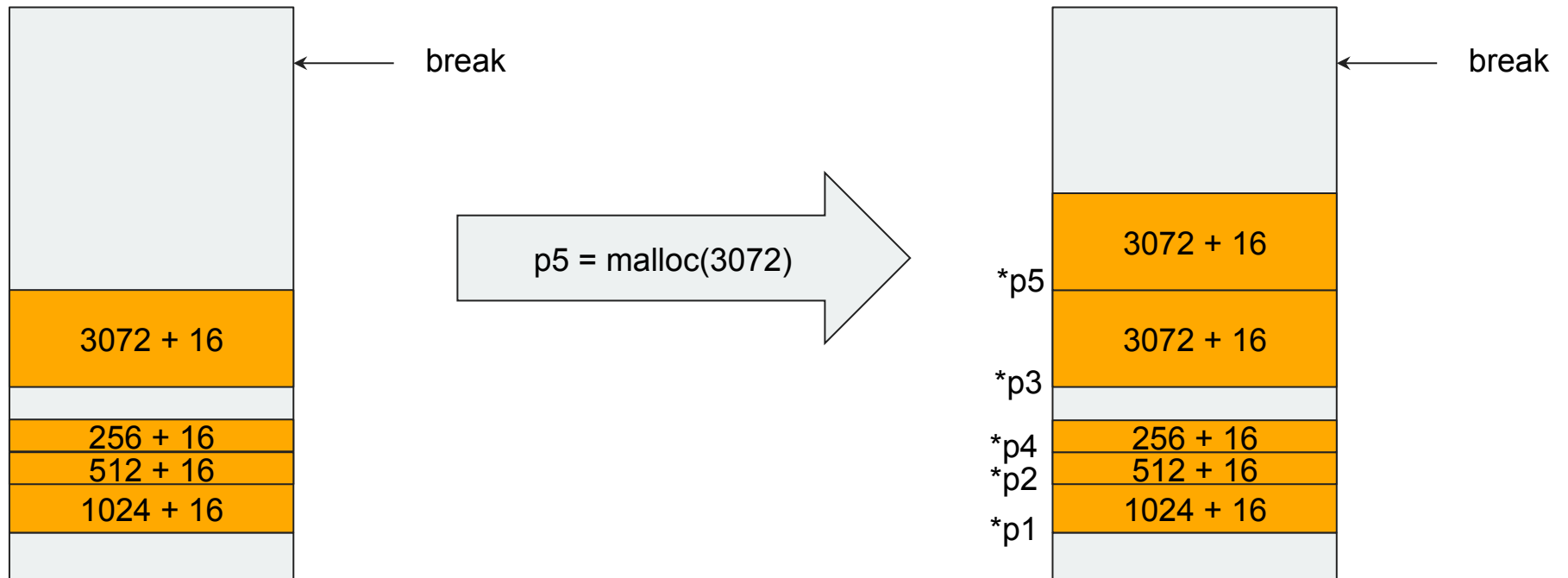
Gestion du tas (heap)



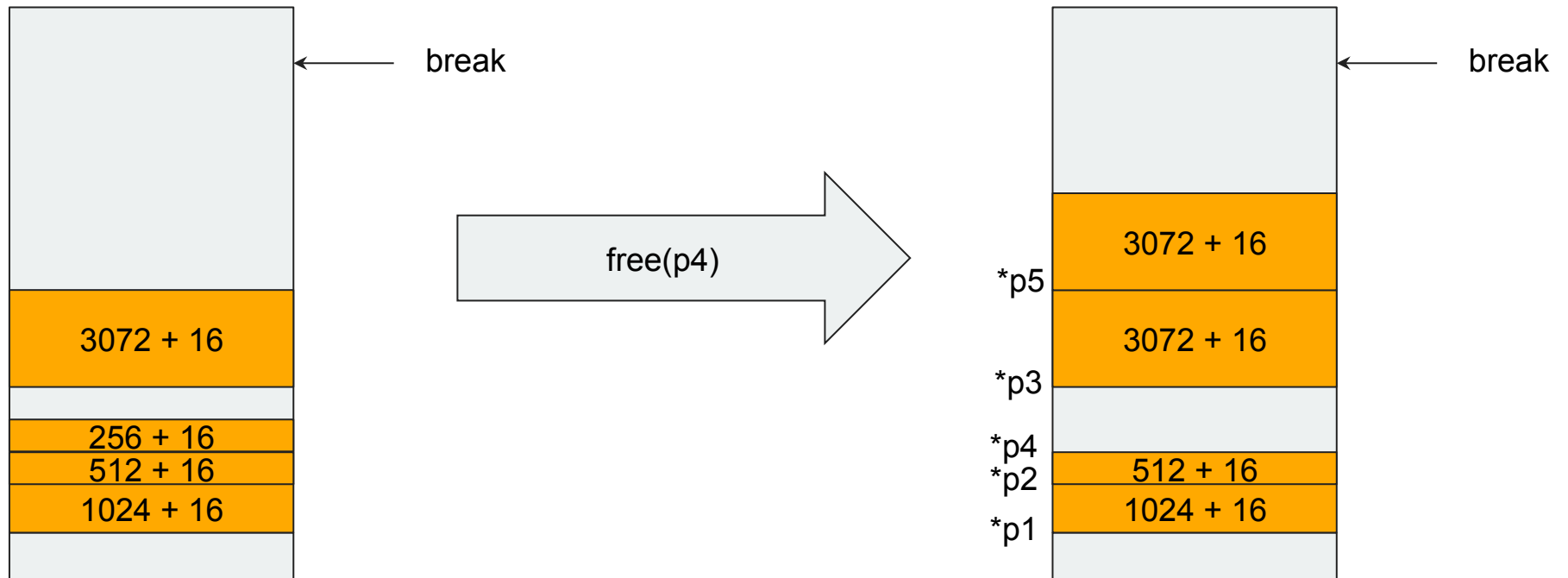
Gestion du tas (heap)



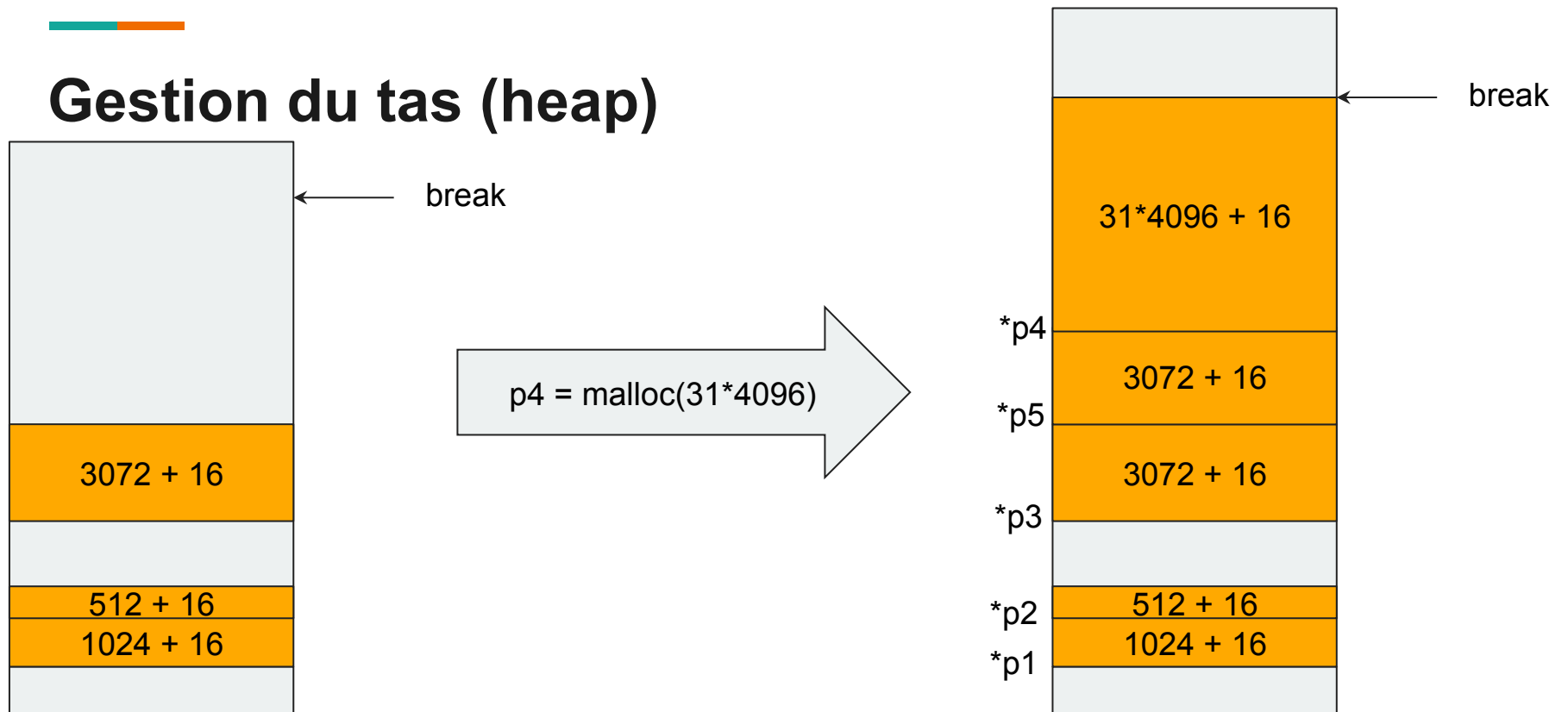
Gestion du tas (heap)



Gestion du tas (heap)



Gestion du tas (heap)





Exemple Gestion de la pile

Testez le code dans le répertoire :
`codes/MemVirt/SegfaultHandler`

- Appels récursifs à une fonction jusqu'à ce qu'une erreur de segmentation se produise (saturation de la pile).

Lecture suggérée

- **Chapitre 6: Gestion de la mémoire**

Introduction aux systèmes d'exploitation - Cours et exercices en GNU/Linux, Hanifa Boucheneb & Juan-Manuel Torres-Moreno, 216 pages, édition ellipses, 2019, ISBN : 9782340029651.

- **Capsules de Vittorio : Composition d'une adresse**

https://www.youtube.com/channel/UCffP7k2AXCttKadZcHsqsYg?view_as=subscriber