Mémoire
Swapping - Adressage Virtuel - Pagination
Programmation Système — R3.05

C. Raïevsky

IUT Valence
Université Grenoble Alpes

Département Informatique
BUT Informatiques 2ème année

2023

Plan

Structuration initiale de la mémoire

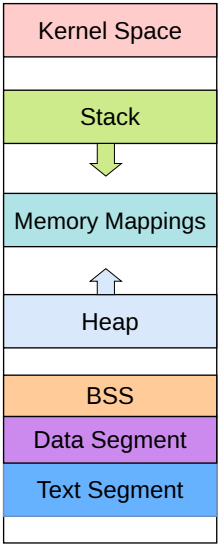
Zones mémoire prédéfinies par l'OS :

"Statiques" :

- Text segment (ro)
- Data segment
- BSS segment

"Dynamiques" :

- Heap (Tas, free-store)
- Stack (Pile)
- Memory Mapping segment



Heap != Stack

Tas – *Heap* ou *Free Store*

- Allocation dynamique de mémoire
- En cours d'exécution
- Explicitement par le programme
- en utilisant `malloc` *et al.*

Pile – *Stack*

- Utilisée pour les appels de fonction
- Pas de manipulation explicite par le programme (du moins en C)
- Instructions processeur dédiées
- Allouée et agrandit par l'OS

Problème d'espace mémoire

Mémoire physique

Épuisement de l'espace mémoire disponible

- Si un processus prend trop de mémoire ou
- s'il y a trop de processus actifs

Solutions

- Va-et-vient – *Swapping*
- Mémoire virtuelle

Va-et-vient – *Swapping*

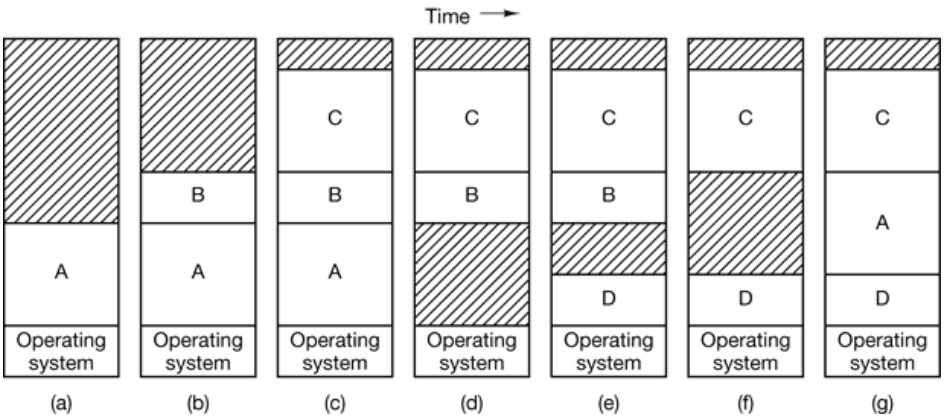
Lorsque un processus est choisi pour s'exécuter, 2 cas possibles :

- Suffisamment de mémoire disponible : chargement en mémoire
- Sinon :
  - déchargement d'un ou plusieurs processus de la mémoire vers le disque dur ;
  - puis chargement du processus choisi

Un processus est :

- soit complètement chargé en mémoire
- soit sur le disque

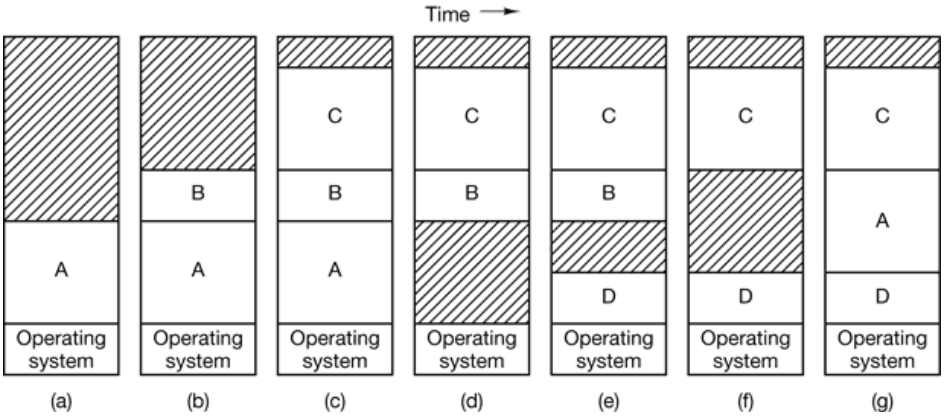
Exemple de séquence d'exécution de 4 processus



Swapping aux transitions :

- (c) → (d) → (e) ⇒ A sort, D rentre
- (e) → (f) → (g) ⇒ B sort, A rentre

Exemple de séquence d'exécution de 4 processus



Comment gérer les espaces libres et occupés ?

- tableau de bits
- liste chaînée

Gestion de la mémoire libre

Tableau de bit – *bitmap*

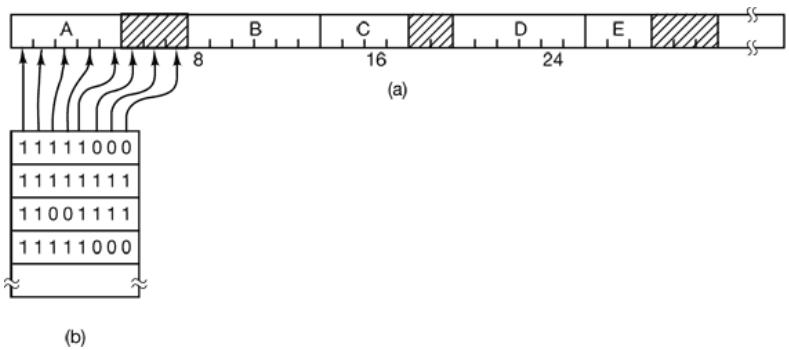


Tableau de bit – *bitmap*

- Mémoire divisée en blocs
- Bit à 1 si le bloc est occupé par un processus
- **Problème** : recherche d'un espace libre longue

Gestion de la mémoire libre

Tableau de bit – *bitmap*

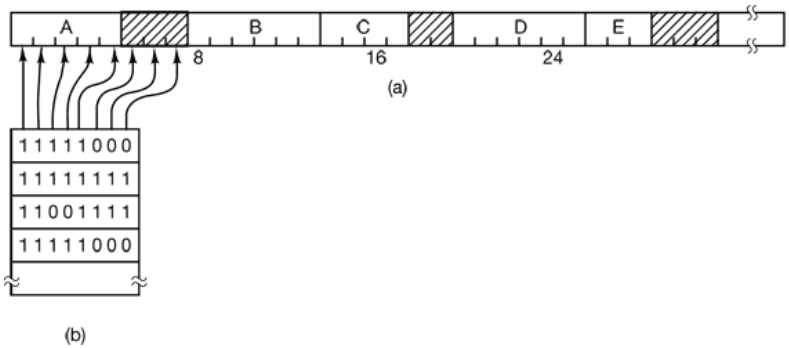
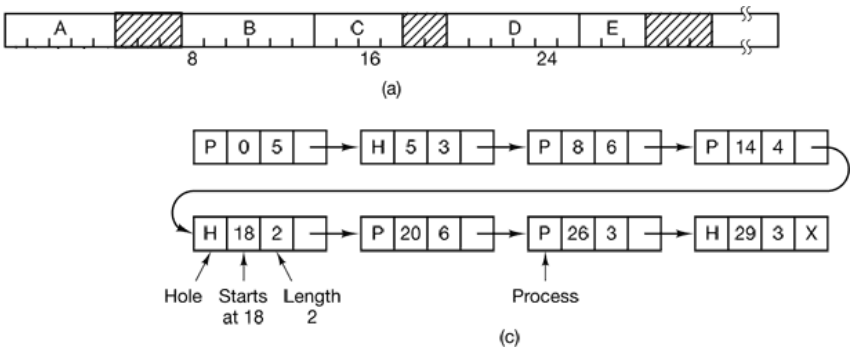


Tableau de bit – *bitmap*

- Mémoire divisée en blocs
- Bit à 1 si le bloc est occupé par un processus
- **Problème** : recherche d'un espace libre longue

Gestion de la mémoire libre

Liste chaînée

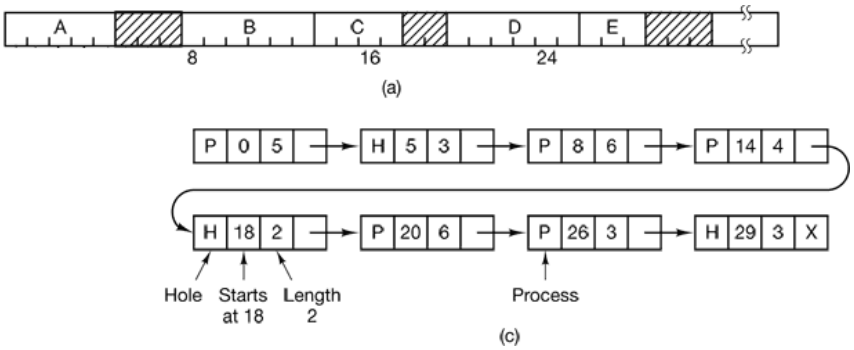


Liste chaînée

- Chaque segment mémoire est représenté par un élément de la liste
- Amélioration de la recherche d'espace libre
- **Un problème persiste** : quel espace libre choisir ?

Gestion de la mémoire libre

Liste chaînée



Liste chaînée

- Chaque segment mémoire est représenté par un élément de la liste
- Amélioration de la recherche d'espace libre
- **Un problème persiste** : quel espace libre choisir ?

Choix d'un espace libre (1/2)

Principaux Algorithmes

Premier espace libre – *First Fit*

- Le premier espace libre suffisamment grand pour contenir le processus est choisi

Espace libre suivant – *Next Fit*

- Le premier espace libre suffisamment grand pour contenir le processus est choisi
- La prochaine recherche commence à l'adresse de cet espace
- Meilleures performances que *first fit* en simulation

Choix d'un espace libre (2/2)

Principaux Algorithmes

Espace libre le plus adéquat – *Best Fit*

- L'espace libre dont la taille est la plus proche de celle demandée est choisi
- Examen de tous les espaces libres à chaque fois
- Entraîne plus de fragmentation que les 2 autres à cause de la sciure – *Saw Dust*

Optimisations

- Utilisation de plusieurs listes – *Quick Fit*
  - suivant le type (occupé, libre)
  - suivant la taille
- Classement des listes par taille pour accélérer la recherche

Problème d'espace, encore...

Cette technique (*swapping*) conserve des problèmes :

- Si un processus prend plus de mémoire que disponible
- Charger et décharger tout un processus prend beaucoup de temps

Solution :

Mémoire virtuelle + pagination

Exemple d'adresses mémoire virtuelles

```c
int g;

int main(int argc, const char *argv[])
{
   printf("&g : [%#010x]\n", &g);
   int j;
   int k;
   printf("&j : [%#010x]\n", &j);
   printf("&k : [%#010x]\n", &k);
   printf("&j-&k: %d\n", &j - &k);

   // Memory allocation on the heap
   int* px1 = malloc(100 * sizeof(int));
   printf("px 1 : [%#010x]\n", px1);

   int* px2 = malloc(42 * sizeof(int));
   printf("px 2 : [%#010x]\n", px2);

   printf("px2 - px1: %d\n", px2 - px1);

   return 0;
}
```

2 exécutions :

```
&g : [0xb670b024]
&j : [0x1453f0dc]
&k : [0x1453f0d8]
&j - &k: 1
px 1 : [0xb6f6f6b0]
px 2 : [0xb6f6f850]
px2 - px1: 104
px1 - &g: 2199971
```

```
&g : [0x29cef024]
&j : [0x080236dc]
&k : [0x080236d8]
&j - &k: 1
px 1 : [0x2a10d6b0]
px 2 : [0x2a10d850]
px2 - px1: 104
px1 - &g: 1079715
```

Exemple d'adresses mémoire virtuelles

Comment être sur que les adresses ne seront pas les mêmes ?

2 exécutions :

```
&g : [0xb670b024]
&j : [0x1453f0dc]
&k : [0x1453f0d8]
&j - &k: 1
px 1 : [0xb6f6f6b0]
px 2 : [0xb6f6f850]
px2 - px1: 104
px1 - &g: 2199971
```

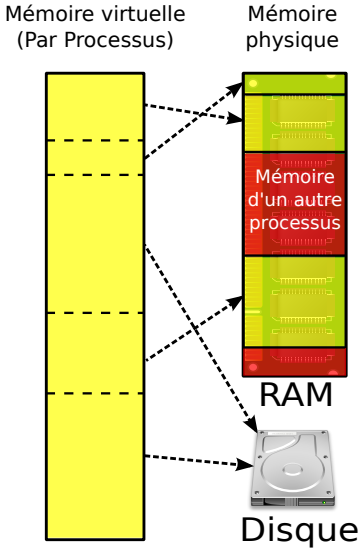
```
&g : [0x29cef024]
&j : [0x080236dc]
&k : [0x080236d8]
&j - &k: 1
px 1 : [0x2a10d6b0]
px 2 : [0x2a10d850]
px2 - px1: 104
px1 - &g: 1079715
```

Adressage mémoire

Mémoire Virtuelle

Chaque processus a un espace d'adressage VIRTUEL

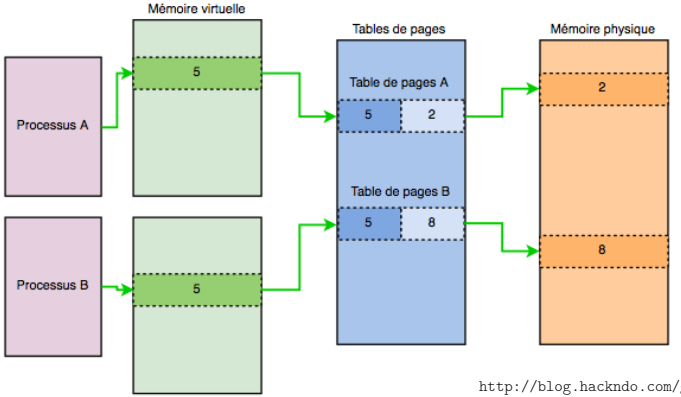
- Évite les conflits d'adresses
- Facilite grandement la multiprogrammation
- Permet de mettre en place des mécanismes de protection
- Nécessite un mécanisme de mise en correspondance



Adressage Mémoire

Correspondance adresse virtuelle ↔ adresse physique

- Mémoire virtuelle et physique découpées en pages
- Chaque processus possède une **TABLE DE PAGE**
- qui contient la correspondance



http://blog.hackndo.com/gestion-de-la-memoire/

Pourquoi des pages ?

Si correspondance directe adresse virtuelle – adresse physique

- Chaque mise en correspondance consomme l'espace d'une adresse virtuelle et l'espace d'une adresse physique
- $\Rightarrow$ La table de correspondance consomme 2/3 de la mémoire...

De nombreux périphériques sont accédés par blocs, qui sont mis en correspondance avec les pages

On parle de :

- "Page" pour les pages virtuelles
- "Cadre de page" – "*Page frame*" pour les pages en mémoire physique

Table des pages

Représentation simpliste

Chaque adresse virtuelle comporte :

- Un numéro de page virtuelle
- Un décalage par rapport au début de cette page (*offset*)

Chaque entrée dans la table des pages comporte :

- Le numéro de page physique auquel elle correspond
- Un indicateur (*flag*) de validité
- Des indicateurs d'accès
  - Est-il possible d'écrire dans cette page ?
  - Contient elle du code exécutable ?

Source : wiki.osdev.org/Paging

Exercice sur la table des pages

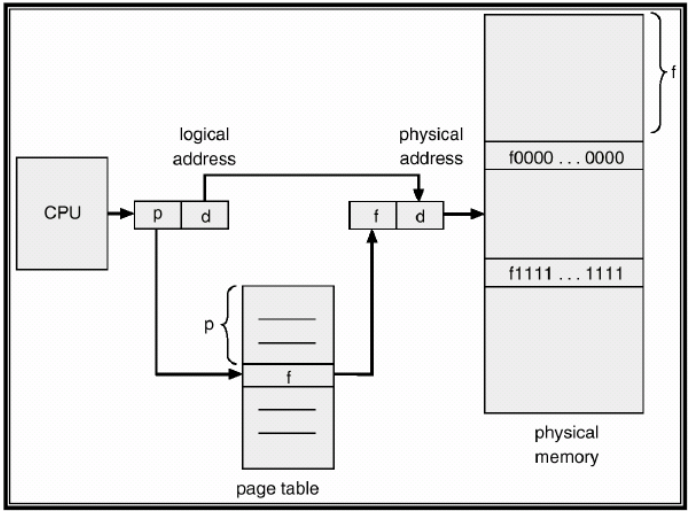
Schématisez un accès mémoire par le cpu utilisant cette représentation

Commencez par :

- Représenter les deux parties de l'adresse virtuelle
- Représenter la table des pages
- Représenter les deux parties de l'adresse physique
- Représenter la mémoire physique
- Faite correspondre les différentes parties des adresses aux différents décalages

Table des pages

Représentation simpliste



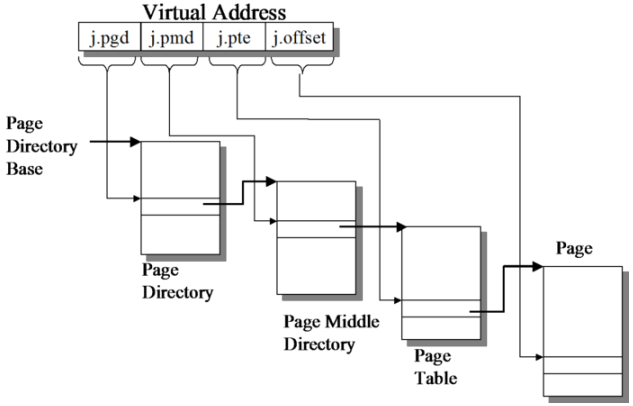
Source : courses.teresco.org/cs432_f02/lectures/11-memory/11-memory.html

Table des pages

Sous Linux

Décomposition hiérarchique des adresses virtuelles

- 3 décalages dans des tables intermédiaires
- Un décalage (*offset*) dans la page correspondante



Défaut de page – *Page fault*

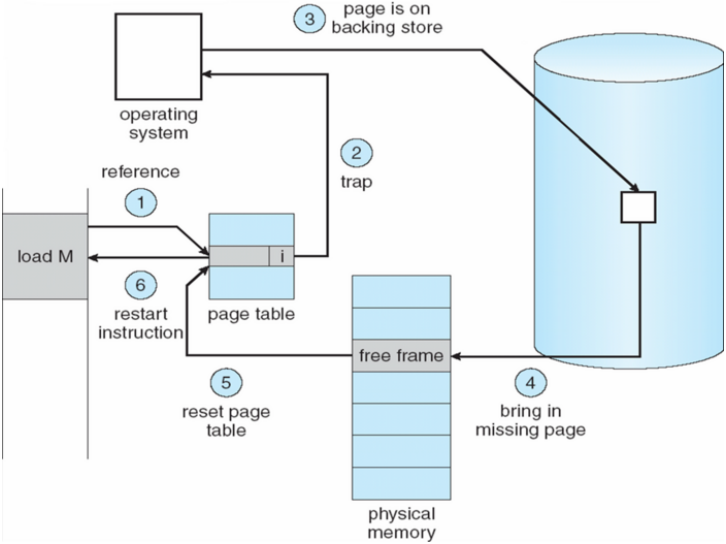
Accès à une page virtuelle qui n'est pas associée à une page physique

Un défaut de page entraîne :

- Une interruption, l'exécution du processus ne peux pas continuer
- Le noyau se charge de
  - charger la page virtuelle accédée
  - décharger au préalable une page virtuelle associée à un cadre de page, **s'il n'y a pas de cadre de page libre**

Un processus qui cause trop de défauts de page fait du "***trashing***"

Défaut de page – *Page fault*



Remplacement de pages

Quelle page décharger en cas de *page fault* ?

The Optimal Page Replacement Algorithm

- Facile à décrire, impossible à implémenter
- $\rightsquigarrow$ Décharger la page qui sera accédée dans le plus longtemps
- Implique de connaître le futur...

Not Recently Used – NRU

- Utilisation de deux bits pour classer les pages :
  - non accédées, non modifiée
  - non accédées, modifiées
  - accédées, non modifiées
  - accédées, modifiées
- remise à zéro du bit "accédée" régulièrement
- Sélection aléatoire dans la classe la plus basse

Remplacement de pages

Quelle page décharger en cas de *page fault* ?

FIFO

- L'OS stocke une liste chaînée des pages chargée
- dans l'ordre de chargement
- la tête de la liste est déchargée en cas de *page fault*

Problème : ne tient pas compte de l'utilisation effective des pages

FIFO avec seconde chance

Identique à FIFO sauf que l'OS examine le bit "accédée" de la page la plus ancienne :

- Si elle n'a **pas** été accédée, elle est déchargée (FIFO)
- Si elle a été accédée,
  - elle est replacée en tête de liste
  - la recherche continue

Remplacement de pages

Quelle page décharger en cas de *page fault* ?

Clock Replacement Algorithm

Idem à FIFO avec seconde chance mais en utilisant une liste circulaire

Least Recently Used

- Déchargement de la page la moins utilisée
- Compliqué et coûteux à mettre en uvre
- Approximations implémentées : *aging*

Working Sets

- Préchargement d'ensembles de pages
- Basé sur une analyse au runtime des ensembles de pages