Gestion de la mémoire

Sacha Krakowiak
Université Joseph Fourier
Projet Sardes (INRIA et IMAG-LSR)

http://sardes.inrialpes.fr/~krakowia

Plan de la présentation

- **Principes de réalisation de la mémoire virtuelle**
  - Définitions et motivations
  - Mécanismes de base
- **Mémoire virtuelle d'un processus**
  - Organisation générale
  - Retour sur *fork* et *exec*
  - Couplage de fichiers : *mmap*
- **Allocation de mémoire**
  - Gestion explicite ou implicite
  - Algorithmes d'allocation
  - Pièges de l'allocation de mémoire

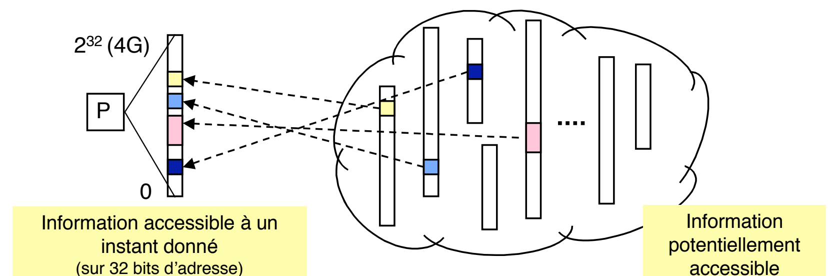
Motivations

Mémoire virtuelle (d'un processus) : ensemble des emplacements accessibles à ce processus (via une adresse dite virtuelle).

- **"Fenêtre" sur l'ensemble des informations accessibles**
  - Un processeur ne peut adresser qu'un espace limité (2^n emplacements, si *n* est le nombre de bits de l'adresse)
- **Gestion économique des ressources**
  - Ne garder en mémoire centrale que les informations nécessaires
  - Permettre le partage d'informations communes
- **Indépendance et protection mutuelle des processus**
  - Chaque processus a sa propre mémoire virtuelle
  - Les processus sont mutuellement protégés contre les erreurs ou les malveillances
  - Néanmoins, possibilité de définir des informations communes (noyau, bibliothèques, données)

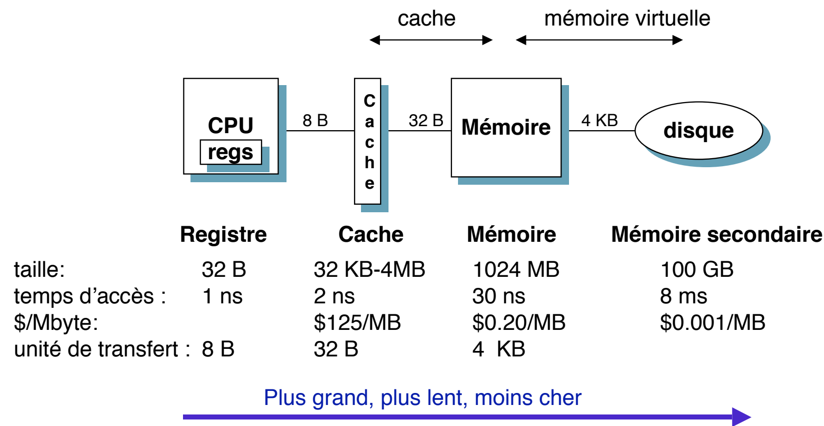
La mémoire virtuelle comme "fenêtre d'adressage"

Le processeur ne peut "voir", à un instant donné, qu'un espace limité par sa capacité d'adressage. Cet espace représente une fenêtre sur l'ensemble de l'information potentiellement accessible.



Deux aspects à considérer :
- **Logique** : définir à tout moment quelles sont les informations accessibles
- **Physique** : permettre l'accès effectif à ces informations

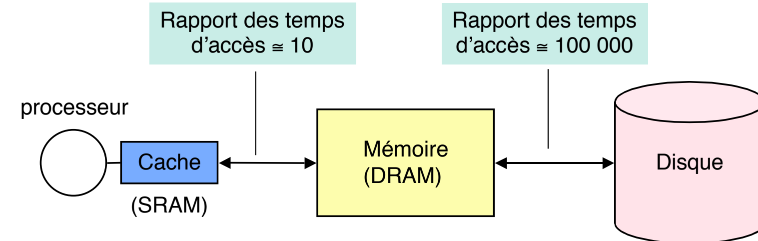
Hiérarchie de mémoire (1)



	Registre	Cache	Mémoire	Mémoire secondaire
taille:	32 B	32 KB-4MB	1024 MB	100 GB
temps d'accès :	1 ns	2 ns	30 ns	8 ms
$/Mbyte:		$125/MB	$0.20/MB	$0.001/MB
unité de transfert :	8 B	32 B	4 KB	

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

Hiérarchie de mémoire (2)



L'essentiel des données est conservé sur disque (taille ≅ x100 fois celle de la mémoire)
Les données sont manipulées en mémoire principale
(le cache est un mécanisme accélérateur)
La bonne gestion des **transferts mémoire-disque** est primordiale pour les performances
La localité des références est importante en raison de la **latence** d'accès au disque
(le transfert du premier octet coûte 100 000 fois plus que celui des suivants

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

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 cases (*frames*) et de pages

Pourquoi une taille fixe ? Commodité de gestion (les cases sont interchangeables)

Comment gérer plusieurs processus indépendants ? Une **mémoire virtuelle** par processus : un ensemble de pages, représentant l'espace adressable par **ce processus**. Les mémoires virtuelles des différents processus sont indépendantes.

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.

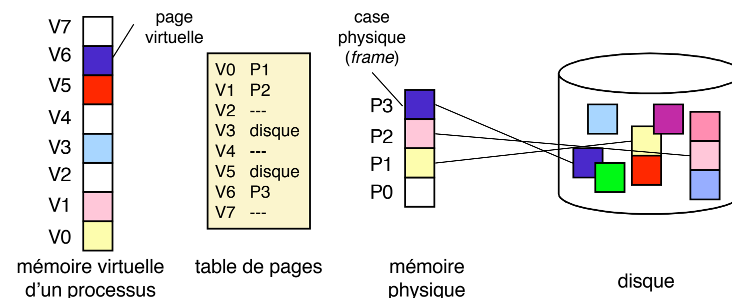
Partage des rôles entre le système d'exploitation et les utilisateurs

- 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.

© 2005-2006, S. Krakowiak

Principe de la mémoire virtuelle paginée (2)

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

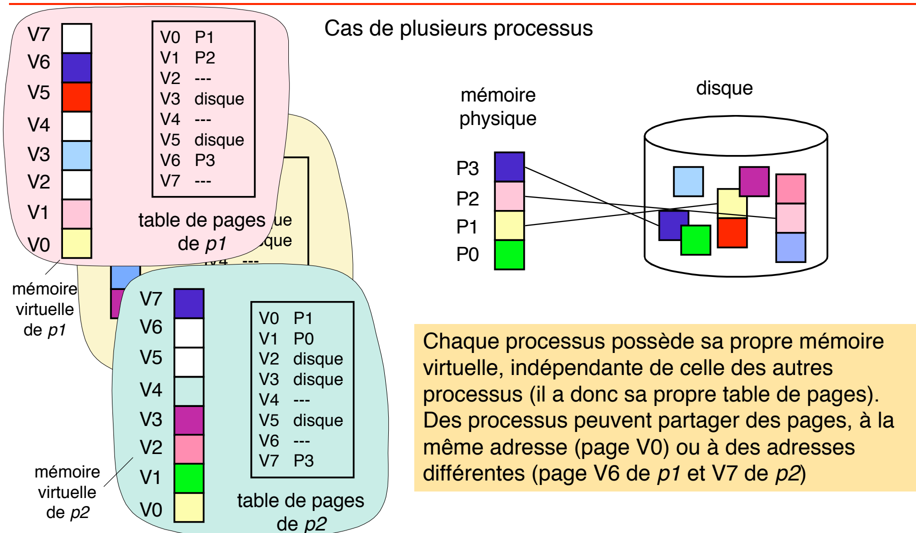


La mémoire virtuelle peut être plus grande que la mémoire physique (mais pas plus grande que la capacité d'adressage du processeur).
Une page virtuelle dont le contenu est en mémoire physique est immédiatement accessible.
Une page virtuelle dont le contenu n'est que sur disque doit être amenée en mémoire physique pour être accessible ; il faut pour cela lui trouver une case libre.
S'il n'y a plus de case libre, il faut en libérer une (donc copier son contenu sur disque s'il a changé).

© 2005-2006, S. Krakowiak

Principe de la mémoire virtuelle paginée (3)

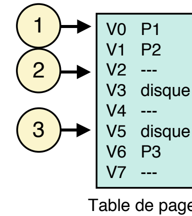
Cas de plusieurs processus



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)



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

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

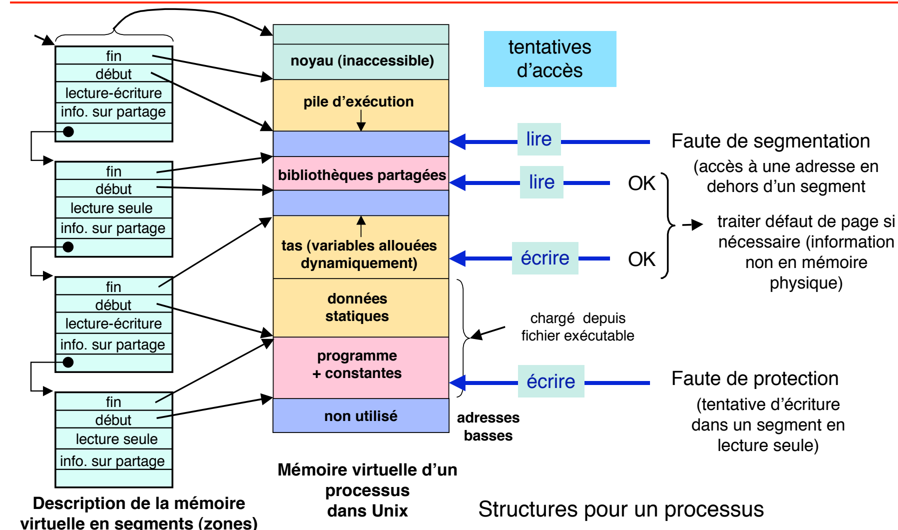
Cas 3 (exemple : V5) : la page correspondante est sur disque (à une adresse indiquée). Il y a défaut de page.

Traitement du défaut de page

Le processus demandeur est bloqué (le processeur est donc alloué à un autre processus). Le système d'exploitation trouve une case 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



tentatives d'accès

lire — Faute de segmentation (accès à une adresse en dehors d'un segment)

lire — OK

écrire — OK

→ traiter défaut de page si nécessaire (information non en mémoire physique)

chargé depuis fichier exécutable

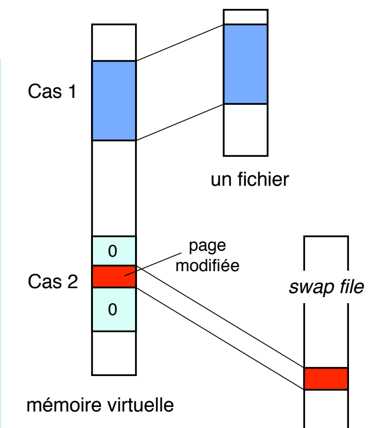
écrire — Faute de protection (tentative d'écriture dans un segment en lecture seule)

Structures pour un processus

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 spécifiées 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.

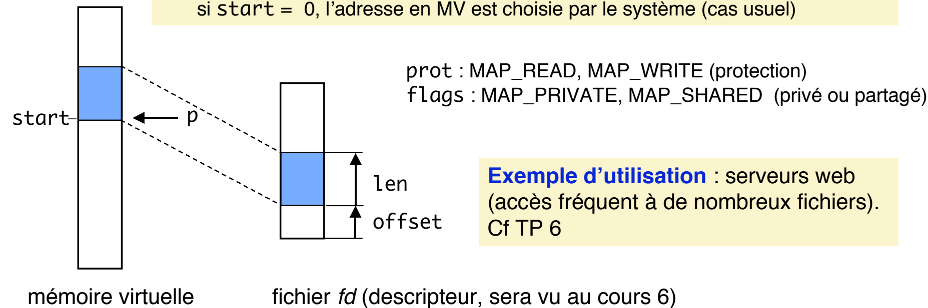


Couplage entre mémoire virtuelle et fichiers

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** (voir plus loin, cours 6). Une page ne sera copiée 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)



`prot` : MAP_READ, MAP_WRITE (protection)
`flags` : MAP_PRIVATE, MAP_SHARED (privé ou partagé)

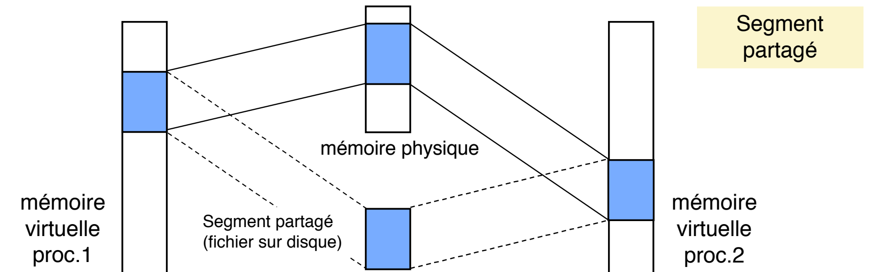
Exemple d'utilisation : serveurs web (accès fréquent à de nombreux fichiers). Cf TP 6

© 2005-2006, S. Krakowiak

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.

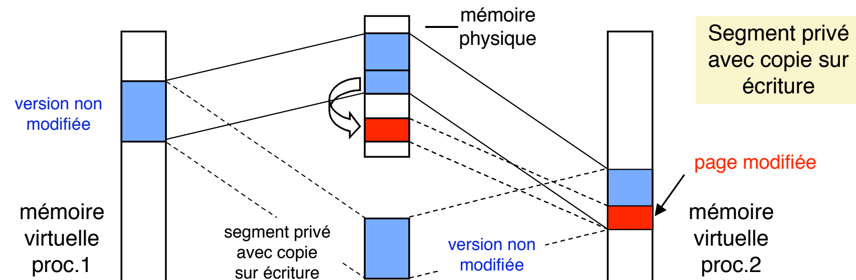
Un segment partagé existe en un seul exemplaire en mémoire physique. Il peut être associé à une zone de la mémoire virtuelle de plusieurs processus. Toute modification par un processus devient visible à tous les autres (et est reportée sur disque).



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.

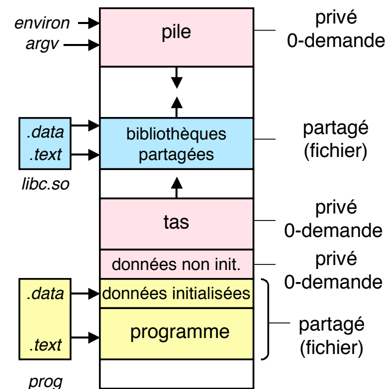
Dans la pratique, chez le fils, *fork()* est généralement suivi d'*exec()*, qui modifie la mémoire virtuelle du fils en y associant un nouveau fichier exécutable.

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 executable), *.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).



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.

© 2005-2006, S. Krakowiak

Allocation dynamique de mémoire : importance

L'allocation dynamique de mémoire est un aspect important

Incidence

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

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

Problèmes

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

En conséquence, l'allocation de mémoire est souvent gérée de manière automatique, invisible aux applications (exemple : Java).

© 2005-2006, S. Krakowiak

Allocation dynamique de mémoire : primitives

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

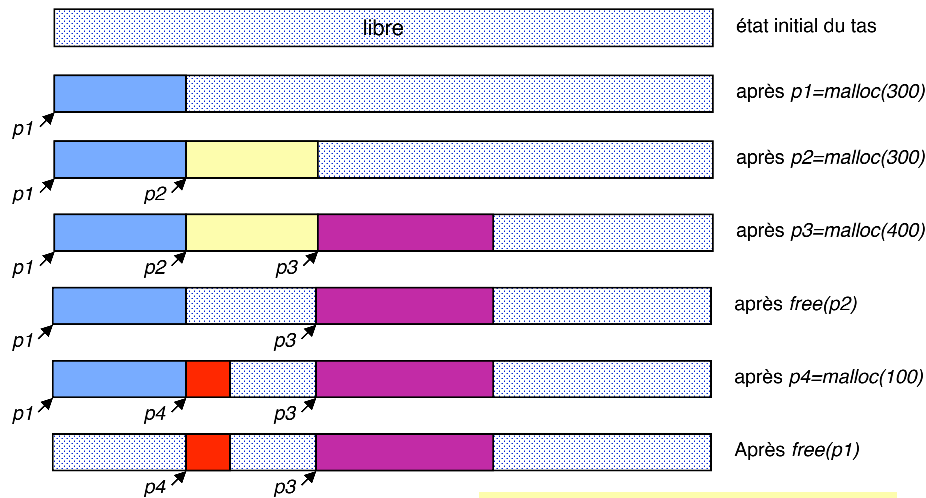
*void *malloc(size_t size)* renvoie un pointeur vers un bloc de mémoire virtuelle de taille au moins égale à *size* (ajusté selon alignement ; en général frontière de double mot : 8 octets). Si erreur (par ex. pas assez de place disponible), *malloc()* renvoie NULL et affecte une valeur au code d'erreur *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.

Dans des cours antérieurs, vous avez vu comment utiliser *malloc()* et *free()* pour gérer des structures dynamiques (tableaux, listes). Dans ce qui suit, on examine la manière de réaliser ces primitives.

© 2005-2006, S. Krakowiak

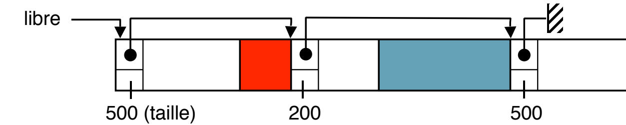
Allocation dynamique de mémoire : exemple introductif



Problèmes : *malloc(600)* ? *malloc(150)* ?

Allocation dynamique de mémoire : algorithmes (1)

Représentation des zones libres : liste chaînée, pointeurs dans zones libres



Pour satisfaire une demande, on parcourt la liste libre, et on peut prendre :

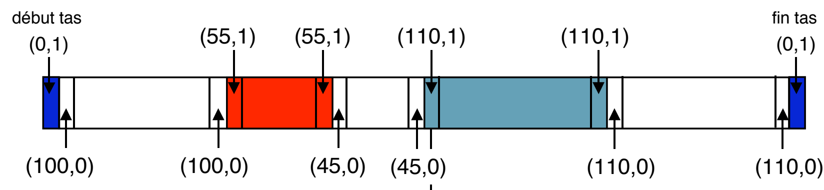
1. La première zone libre qui convient (qui est assez grande pour satisfaire la demande (***first fit***). Avantage : rapidité. Inconvénient : risque de mauvaise utilisation de l'espace libre (perte de grands segments)

   Variante usuelle (***next fit***) : la recherche commence à partir de la position courante du pointeur de parcours (et non à la première zone libre). Avantage : évite l'accumulation de petites zones libres en début de liste.

2. La zone dont la taille est plus proche de la demande (***best fit***). Avantage : meilleur ajustement. Inconvénients : plus lent ; risque de créer de nombreuses zones de faible taille (émiettement).

Allocation dynamique de mémoire : algorithmes (2)

Autre représentation des zones libres : marqueurs de début et de fin



Chaque zone (libre ou allouée) comporte deux marqueurs identiques, au début et à la fin (l'espace utile exclut les marqueurs).

Un marqueur contient la taille de la zone et un bit d'occupation (0 : libre, 1 occupé)

Avantage : il est facile de réunir deux zones libres adjacentes lors d'une libération : 2 marqueurs voisins avec bit d'occupation à 0).

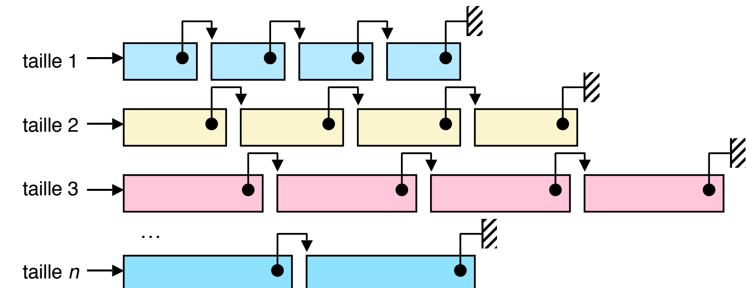
Les mêmes politiques de recherche (*first fit*, *next fit*, *best fit*, ou autres) sont utilisables.

Allocation dynamique de mémoire : algorithmes (3)

Très souvent, la demande porte sur un nombre assez restreint de tailles différentes (déterminées après observation des applications).

On organise la mémoire en plusieurs pools, chacun constitué de zones d'une taille donnée. On peut prévoir un pool spécial pour les demandes hors des tailles standard, ou arrondir à la taille supérieure la plus voisine.

Cette méthode est souvent adoptée, car elle concilie efficacité et bonne gestion de l'espace libre.



Allocation dynamique de mémoire : critères

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

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

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

© 2005-2006, S. Krakowiak

Pièges de l'allocation dynamique de mémoire (1)

Mauvais déréférencement de pointeurs

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

où est l'erreur ?

Mémoire non initialisée

```
/* calcule A*x (matrice*vecteur) */
#define N ...
int *matvec (int **A, int *x) {
	int y = malloc(N*sizeof(int));
	int i, j;
	for (i=0; i<N; i++)
		for (i=0; i<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*, Addison-Wesley, 2003

Pièges de l'allocation dynamique de mémoire (2)

Allocation erronée

```
/* 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 ?

Débordement de tampon

```
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*, Addison-Wesley, 2003

Pièges de l'allocation dynamique de mémoire (3)

Arithmétique des pointeurs

```
/* 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 ?

Fuite de mémoire

```
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*, Addison-Wesley, 2003

Résumé de la séance 5

- **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**
  - **Compléménts sur *fork()* et *exec()* ; copie sur écriture**
  - **Couplage de fichiers : *mmap()***
- **Allocation dynamique de mémoire**
  - **Primitives *malloc()* et *free()* ; techniques de réalisation**
  - **Pièges de l'allocation dynamique de mémoire**

© 2005-2006, S. Krakowiak