La mémoire

Polytech Paris-Sud

Cycle ingénieur de la filière étudiant

Louis Mandel

Université Paris-Sud 11

Louis.Mandel@lri.fr

année 2009/2010

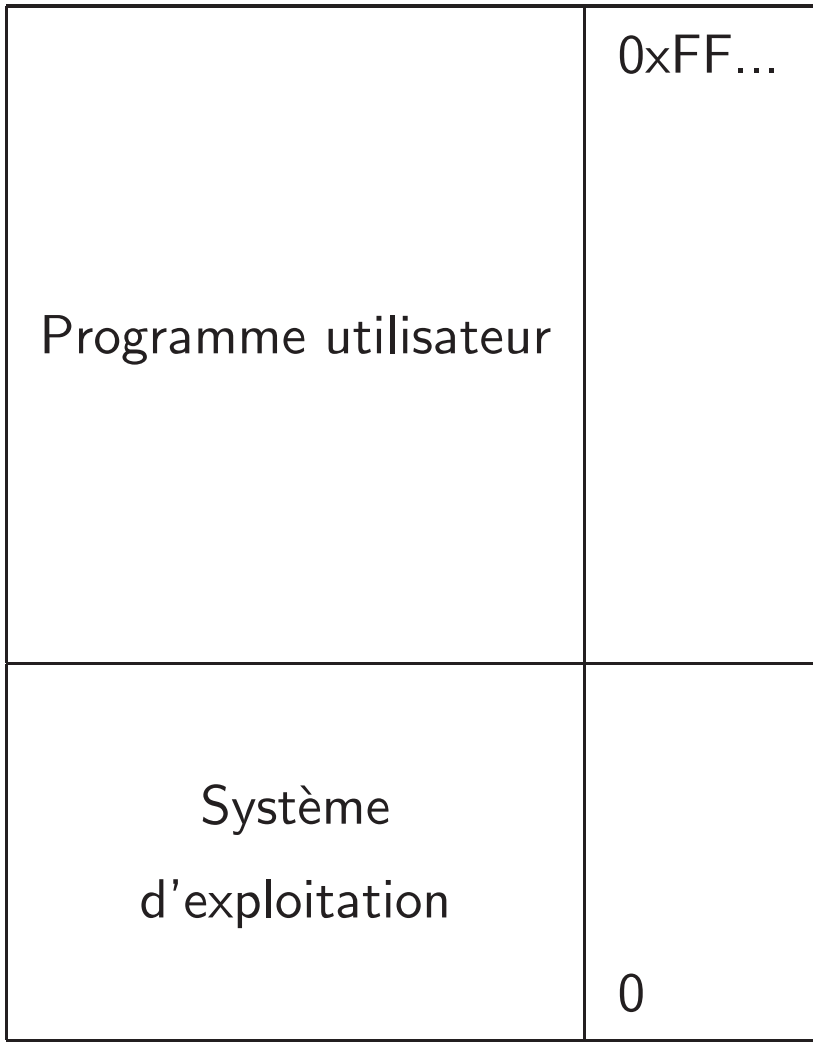
Aspects matériels de la mémoire

Types de mémoires

Type	Accès	Vitesse	Persistance	Domaine d'utilisation
Registre	lecture-écriture	*****	non	temporaire uniquement
SRAM	lecture-écriture	****	non	buffer
DRAM	lecture-écriture	***	non	mémoire centrale
ROM	lecture	**	oui	mémoire de démarrage (BIOS)
SWAP	lecture-écriture	*	oui	mémoire virtuelle

Gestion élémentaire de la mémoire

- Un système d'exploitation et un processus



Mémoire et multiprogrammation

Mémoire et multiprogrammation

- Situation :
  - Nombre arbitraire de processus
  - Plusieurs processus « simultanément » en mémoire
  - Une seule mémoire pour tout le monde
  - Usage dynamique de la mémoire
- Attentes :
  - Sécurité : seul le processus propriétaire d'une zone mémoire peut en lire le contenu
  - Intégrité : un processus ne peut modifier (volontairement ou non) la mémoire d'un autre processus
  - Disponibilité : le système doit satisfaire un maximum de demandes de mémoire.

Mémoire et multiprogrammation : Solutions

- Échanges superposés
  - L'image mémoire du processus devenant actif est recopiée depuis le disque
- Partitions fixes de la mémoire
  - la mémoire est découpée en zones de tailles fixes
  - Problèmes
    - Fragmentation interne : certains processus peuvent ne pas utiliser toutes la mémoire qui leur est allouée
    - Fragmentation externe : ils se peut qu'il n'y est pas de segment continue de mémoire pouvant accueillir un processus alors que la somme de l'espace dans les zones libres serait suffisant

Mémoire et multiprogrammation : Solutions

- Partitions variables et va-et-vient (*swap*)
  - Les partition sont allouées selon les besoins des processus
  - La taille des partitions peut changer dynamiquement
  - Les programmes doivent être relogeables
  - Gestion de la mémoire
    - par bitmap d'occupation
    - par liste chaînée
  - Limites :
    - Un processus doit résider en totalité en mémoire
    - Problème de réallocation de ressources

Mémoire et multiprogrammation : Solutions

- Mémoire virtuelle et pagination
  - L'espace d'adressage virtuel est décomposé en pages logiques
  - Les pages qui contiennent des données sont associées à de la mémoire réelle
  - Un mécanisme matériel traduit les adresses virtuelles en adresses physiques

Mémoire virtuelle et pagination

- Avantages :
  - Les programmes n'ont plus besoin d'être relogeables
  - Les pages n'ont pas l'obligation d'être contiguës (réduit la fragmentation)
  - L'espace total d'adressage peut dépasser la taille de la mémoire physique
- Exemple : Architecture 32 bits, pages de 2^{12} octets (4 Ko)
  - Espace d'adressage $= 2^{32}$ octets $=$ 4 Go
  - 1 048 576 (2^{20}) pages

Mémoire virtuelle et pagination : gestion Matérielle

- La traduction entre adresse virtuelle et adresse réelle est effectuée à chaque accès mémoire
- Les adresses sont composées d'un numéro de page et d'un décalage dans la page
- La mémoire réelle en cadres de la même taille que les pages logiques
- L'unité de gestion mémoire ou MMU (*Memory Management Unit*) traduit les adresses virtuelles en adresses réelles
- La MMU doit :
  - savoir quelles pages virtuelles sont chargées en mémoire et où elles se trouvent
  - pouvoir communiquer avec le système d'exploitation pour prévenir s'il veut accéder à une page qui n'est pas en mémoire (défaut de page)
- Le système doit pouvoir traiter les défaut de pages

Mémoire virtuelle et pagination : Remarque

- Problème de la taille de la table de translation des adresses
  - Table à plusieurs niveaux
  - Utilisation de cache de pagination : TLB (*Translations Lookaside Buffers*)

Défaut de page

- Le mécanisme de mémoire virtuelle permet d'adresser plus de pages qu'il n'y a de cadres.
  - Un certain nombre de pages sont donc déplacer sur le disque.
  - Même si le nombre de cadre est suffisant les systèmes tendent à basculer certains pages sur le disque pour augmenter la taille des caches de disque.
- Lors d'un défaut de page, ou lorsqu'une nouvelle page est allouée et qu'il n'y a plus de cadres disponibles une page est choisis pour être envoyée sur le disque.

Algorithme optimal

- Lorsqu'une page sur le disque a besoin d'être échangée avec une page en mémoire, alors :
  - regarder toutes les pages en mémoire
  - trier sur le critère : « va être utilisée au temps T » avec T croissant
  - choisir celle en fin de liste
  - échanger cette page avec celle sur le disque

Algorithme optimal

- Problème :
  - Il suppose une connaissance de toute l'exécution, c'est à dire toutes les références mémoires qui seront effectuées.
- Algorithme irréalisable
- Par contre l'algorithme peut être utiliser comme référence pour évaluer un algorithme de remplacement :
  - On fait tourner une simulation du processus pour tracer chaque opération mémoire
  - On applique l'algorithme optimal sur la trace obtenue
  - On peut maintenant comparer les défauts de pages avec l'algorithme à évaluer.

Choix aléatoire

- Choix aléatoire de la page à remplacer
- Inconvénient : peut supprimer des pages utiles
- Avantage : n'a pas d'effets pervers si le choix est suffisamment aléatoire
- Pas si mauvais en pratique

NRU : Not Recently Used

- On utilise deux bits d'information pour chaque page :
  - R : la page a été lue
  - M : la page a été modifiée
- Au lancement du processus toutes les pages ont R et M à 0.
- Régulièrement R est repassé à 0 pour toutes les pages.
- Lors d'un défaut de page le système classe les pages dans quatre ensembles :
  1. Ni utilisée ni modifiée
  2. pas utilisée, mais modifiée
  3. utilisée, mais pas modifiée
  4. utilisée et modifiée
- NRU remplace une page au hasard dans le premier ensemble non vide.
- Il est facile à implanter, et donne des performances acceptables.

FIFO : First In First Out

- Simple chaînage des pages selon leur âge
- Algorithme :
  - Lorsqu'une page arrive, elle vient remplacer la plus vieille
  - et deviens la plus jeune
- Inconvénients :
  - ne prend pas en compte « l'utilité » des pages
  - Anomalie de Belady : on peut dans certains cas augmenter les échanges nécessaires si on augmente la taille de la mémoire.

Second Chance Page Replacement

- Amélioration du FIFO : utilisation du bit R
- Régulièrement, les bits R de toutes les pages sont mis à 0.
- Lors du remplacement de page :
  - ▷ On sélectionne la page en queue de la liste (la plus vieille)
  - ▷ Si R vaut zéro, la page est vieille ET non utilisée récemment
  - ▷ Si R vaut un, la page est vieille MAIS TOUJOURS utilisée : elle est remise en tête de la liste (elle devient jeune), et la recherche continue

- Question : cet algorithme termine-t-il ? Quel est son comportement si toutes les pages sont utilisées ?

Clock Page Replacement

- Bien que l'algorithme précédent soit efficace, il passe son temps à déplacer des éléments dans la file
- On peut l'implanter de manière beaucoup plus maligne avec une liste circulaire, et un pointeur sur la position actuelle dans la liste
- Seul ce pointeur se déplace.
- Il s'appelle algorithme d'horloge car les pages peuvent représenter les heures d'une horloge et le pointeur une aiguille

Least Recently Used

- LRU : remplace la page la moins récemment utilisée
- Nécessite un tri dynamique des pages par ordre d'accès à chaque référence mémoire
- C'est possible, en utilisant les bits de protection, mais extrêmement coûteux
- Certaines machines fournissent des compteurs matériels qui permettent de les implanter, mais là aussi, c'est assez coûteux.
- On cherche donc à approximer le même résultat qu'un LRU, sans faire d'action spécifique à chaque accès mémoire.
- Remarque : Situation dégénérée lors de l'accès circulaire à N+1 pages dans une mémoire de N cadres

Not Frequently Used

- L'idée de NFU est d'essayer de simuler NRU :
  1. régulièrement, le système parcours toutes les pages et incrémente un compteur pour toutes les pages dont le bit R vaut 1
  2. et met le bit R à zéro
  3. Ce compteur est une approximation de l'intérêt que porte le processeur à une page (LRU)
- Problème du NFU : il n'oublie rien, et si une page a été souvent utilisée dans l'histoire, elle peut conserver longtemps un compteur supérieur à une page fréquemment utilisée maintenant.

Aging

- Aging est un algorithme qui se base sur NFU
- Avant d'ajouter la valeur du bit R au compteur, on divise le comptuer par 2 (on le décale d'un bit sur la droite)
- et le bit R est ajouté au bit de poids fort du compteur, pas au bit de poids faible
- Une page qui n'a pas été référencée depuis 4 périodes a ses 4 premiers bits à zéro, alors qu'une page qui a été référencée il y a 3 périodes a son 3ième bit de poids fort à 1.
- En prenant la page avec le compteur le plus faible, on supprime la page la moins utilisée (avec l'approximation du temps)

Aging vs LRU

- Aging est moins précis que LRU à cause de l'échantillonnage du temps. Si deux pages n'ont pas été référencées depuis 2 périodes, mais l'ont été il y a trois périodes, Aging ne fait pas la différence, alors que LRU choisit celle qui a été référencée la première il y a trois périodes
- Aging fonctionne en mémoire bornée, et cette mémoire représente la longueur de son histoire. Si deux pages ont un compteur à 0 sur 8 bits, c'est qu'elles n'ont pas été utilisées depuis 8 périodes de temps. Le fait que l'une ait été utilisée il y a 9 périodes, et l'autre il y a 1024 périodes ne se voit pas dans Aging, mais se voit dans LRU

Working Set Page Replacement

- L'idée : un processus utilise un petit nombre de pages à un instant donné
  - ▷ C'est ce qu'on appelle l'ensemble de travail d'un processus.
- Lorsqu'un processus est ordonnancé il est souvent valable de charger simultanément toutes les pages de son ensemble de travail.
- L'algorithme de remplacement de pages essaye de construire ces ensembles, et considère comme candidat à l'éjection une page qui ne fait pas partie d'un ensemble de travail.
  - ▷ On ne construit pas exactement ces ensembles.
  - ▷ On considère que si une page n'est pas utilisée pendant τ périodes de temps, c'est qu'elle ne fait plus partie de l'ensemble de travail.

Working Set Page Replacement : algorithme

- Chaque entrée de la table des pages maintient une date logique entière (numéro de période de dernière utilisation)
- A chaque période de temps, on parcours toute la table
  - Si R = 1, on note le numéro de la période dans le champs de l'entrée
  - R est mis à 0
- lorsqu'un défaut de page arrive, on parcours la table des pages
  - Si R = 0, la page est candidate à l'éjection.
    - Si sa date de dernière référence est plus grande que τ, elle ne fait pas partie de l'ensemble et est éjectée.
    - Sinon, on conserve sa date de dernière utilisation et on continue le parcours
  - Si à la fin de parcours on n'a pas trouvé de page en dehors de l'ensemble, on utilise la plus vieille

WSClock Page Replacement

- Structure de données : liste circulaire de cadres + pointeur (clock)
- Chaque entrée contient la date de dernière utilisation et le bit R.
- A chaque défaut de page, on examine l'entrée pointée en premier
  - ▷ Si le bit R $= 1$, on copie la page sur le disque, le bit R est mis à 0 et le pointeur avance. L'algorithme continue sur la nouvelle page.
  - ▷ Si le bit R $= 0$ et la date est supérieure à τ, la page est inutilisée et on en a une copie propre sur le disque. Donc on la remplace.
- De par sa simplicité et son efficacité, cet algorithme est souvent celui qui est implanté

Comparaison

Optimal	non implantable
Aléatoire	Trop simple
NRU	Très simple
FIFO	Peu facilement supprimer des pages utiles
Second Chance	Grand amélioration de FIFO
Clock	Implantation efficace de second chance
LRU	Excellent mais difficile à implanter
NFU	Approximation simpliste de LRU
Aging	bonne approximation de LRU
Working set	implantation coûteuse
WS clock	meilleur approximation de LRU, implantation efficace

Mémoire virtuelle et segments

Mémoire virtuelle et segments

- Chaque processus est exécuté comme s'il avait toute la mémoire
- Un processus peut accéder uniquement à sa mémoire
- La mémoire virtuelle est décomposée en segments
- Facilite le partage et la protection
- Typiquement, il y a les segments suivants :
  - Code (ou texte)
  - Données statiques
  - Tas (*heap*)
  - Pile (*stack*)

Mémoire virtuelle

- Le segment de code
  - Chargement en mémoire de l'exécutable (Linux : format ELF)
- Le segment de données
  - Variables globales
  - Variables locales `static`
  -
```
#include <stdio.h>
int i;
int j=42;
void count() {
  static int cpt = 0;
  printf("%d\n",++cpt);
}
```

Mémoire virtuelle

- La pile
  - Variables locales, paramètres de fonctions
  - Allouées et desallouées « automatiquement » (pas de `free`)
  - ```
    int *illegal_reference() {
      int i;
      return (&i); /* illegal */
    }
    ```
  - Allocation en pile : `void *alloca(size_t size);`
  - ```
    void stack_alloc(int n) {
      int *array;
      array = alloca(n * sizeof(int));
      return;
    }
    ```

Mémoire virtuelle

- Le tas
  - Gestion dynamique de la mémoire
  - `malloc`, `calloc`, `free`, etc.
  - Exemple :

```
#include <stdlib.h>
void heap_alloc(int n) {
  int *array1, *array2;
  array1 = (int *) malloc(n * sizeof(int));
  array2 = (int *) calloc(n * sizeof(int));
  ...
  free(array1);
  free(array2);
  return;
}
```

Gestion du tas

Gestion du tas

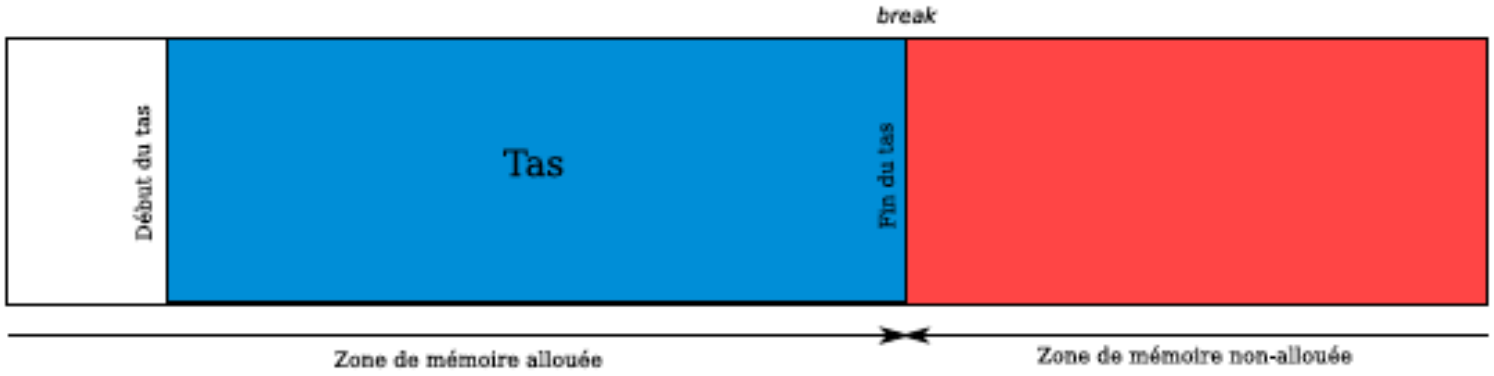
- Changement de la taille du tas
  - Déplacement absolu :

    ```
    int brk(void *end_data_segment);
    ```

  - Déplacement relatif

    ```
    void *sbrk(intptr_t increment);
    ```

- Ces deux fonctions ne sont pas pour « l'interface programmeur », elles sont utilisées seulement pour le développement du noyau.

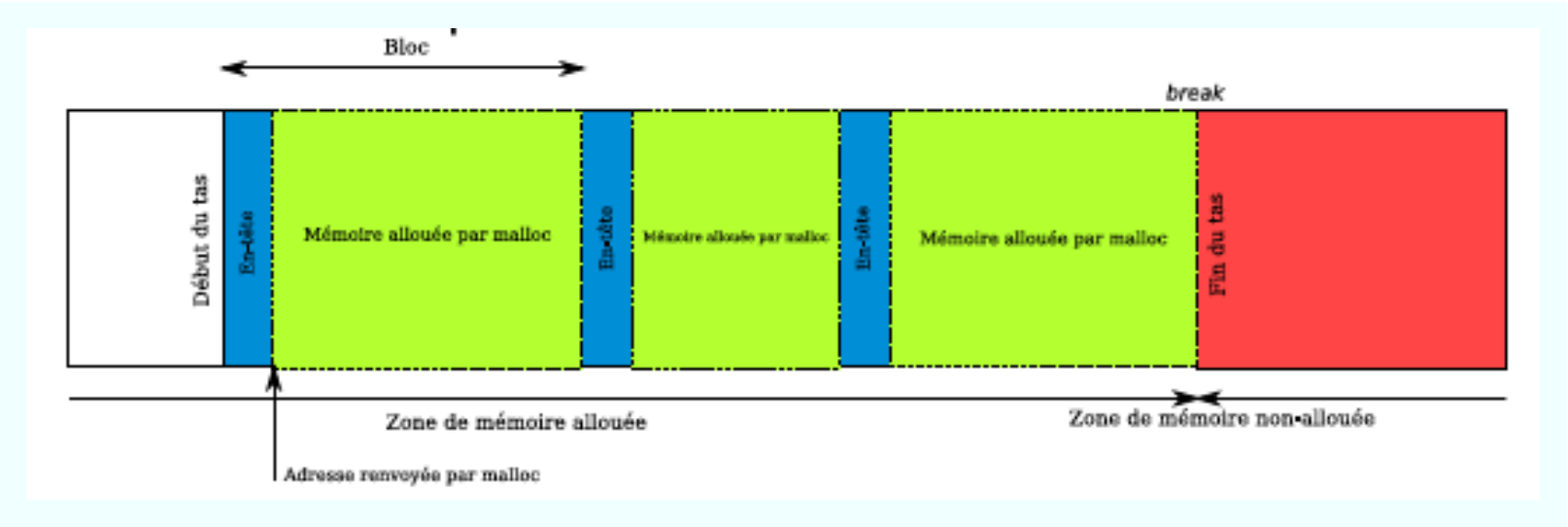


Gestion du tas : `malloc`

- La fonction `malloc` de la bibliothèque standard de C fournit une interface simple pour allouer des blocs de mémoire.
- `void *malloc(size_t size);`
- On attend les propriétés suivantes de `malloc` :
  - La distance entre l'adresse retournée et le break est supérieure à la taille demandée ;
  - Aucun autres appels à `malloc` n'allouera une zone de mémoire déjà allouée ;
  - `malloc` cherche à minimiser la différence entre la quantité totale de mémoire allouée et la quantité réellement utilisée ;
  - Le coup en temps de `malloc` doit être le plus petit possible.

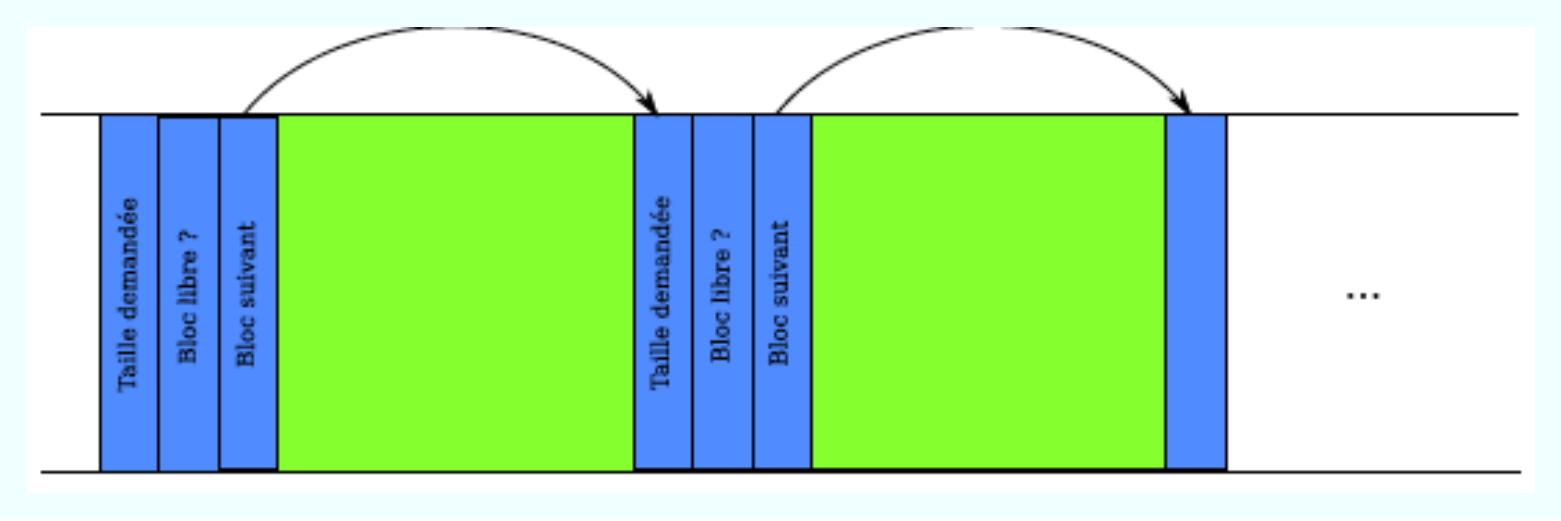
Gestion du tas : `malloc`

- `malloc` structure le tas en blocs.
  - Chaque bloc commence par un en-tête contenant les informations nécessaires au bon fonctionnement de `malloc`
  - le reste correspond à la mémoire fournit par l'interface.



`malloc` **simple**

- L'en-tête est utilisé pour réaliser un chaînage des blocs
- Il contient trois informations :
  - la taille du bloc (celle demandée)
  - un indicateur pour savoir si le bloc est utilisé
  - l'adresse du prochain bloc

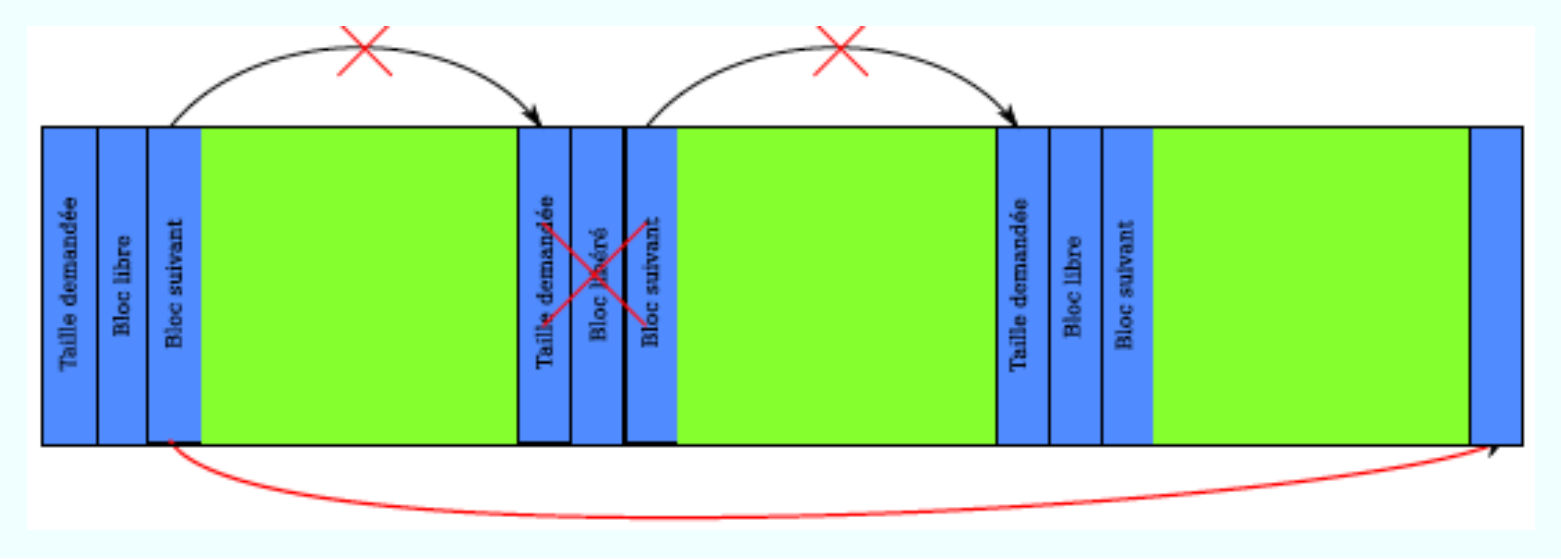


`malloc` simple

- `malloc` (dans sa version initiale) est basé sur un algorithme dit de *first fit*.
- Voici l'idée générale du fonctionnement de `malloc` :
  - Au premier appel :
    - L'adresse du `break` est sauvée dans `base`
    - `break` est décalée de la taille demandée plus la taille de l'en-tête dont les champs sont renseignés
    - La valeur de retour est (`base` + taille de l'en-tête)
  - En dehors du premier appel : l'opération change en fonction de la recherche du premier bloc libre de taille suffisante :
    - En cas d'échec : l'opération est similaire au premier appel.
    - Sinon : le bloc trouvé est divisé pour correspondre à la taille demandée. Si l'espace restant est suffisant, un nouveau bloc est créé.

`free` et fusion

- L'opération de libération de la mémoire est relativement simple :
  - Trouver le bloc
  - Passer l'indicateur d'occupation sur libre
- Cette méthode est très rapide, mais entraîne une importante fragmentation.
  - Une solution consiste à fusionner le bloc libéré avec ses voisins s'ils sont libres.



`free` et fusion : double chaînage

- Pour réaliser la fusion simplement, il faut utiliser un double chaînage : l'en-tête contient un champ avec l'adresse du bloc précédent.
- La structure représentant l'en-tête devient :

```
struct block {
  size_t size ;
  struct block *next ;
  struct block *prev ;
  int used ;
}
```

Amélioration de `malloc`

- Quelques problèmes de l'algorithme de `malloc` présenté :
  - beaucoup de fragmentation
  - temps de recherche d'un bloc libre
- Pistes pour résoudre ces problèmes :
  - Usage d'un algorithme de *best fit*. On recherche le meilleur bloc libre (celui dont la taille est la plus proche).
  - Indexation séparée des blocs libres et utilisées.
  - Listes triés de taille de blocs :
    - Le début du tas contient un tableau de pointeur.
    - Chaque pointeur correspond au début d'une liste de bloc de même taille.
    - Le tableau est trié en fonction de la taille des blocs.
  - Séparation des blocs et des en-têtes (minimise les défauts de pages).