Module NOY

Allocation dynamique de mémoire

- Algorithmes d'allocation
- Ramasse-miettes

istic Informatique Électronique

Allocation dynamique de mémoire

- Objectif
  - Demande de mémoire supplémentaire à l'exécution
  - Tailles et durées d'utilisation des zones de mémoire quelconques
- Interface typique
  - *void *malloc(size_t size)* : demande d'une zone de mémoire de taille *size* et retour de son adresse
  - *void free(void *ptr)* : libération d'une zone de mémoire allouée au préalable (rq : on ne passe pas la taille en paramètre)

Allocation dynamique de mémoire

- Domaines d'utilisation
  - Systèmes sans pagination : allocation de mémoire réelle
  - Systèmes avec pagination : allocation de zones dans l'espace d'adressage virtuel utilisateur, allocation en mémoire physique pour le système d'exploitation

Allocation dynamique de mémoire Terminologie

- Zone : suite d'emplacements mémoire contigus, de taille non fixée a priori
- Zone caractérisée par son adresse de début et sa taille
- Zone libre (trou) : zone de mémoire non allouée par le système
- Zone occupée : partie de mémoire allouée à un processus

Allocation dynamique de mémoire
Problèmes à résoudre

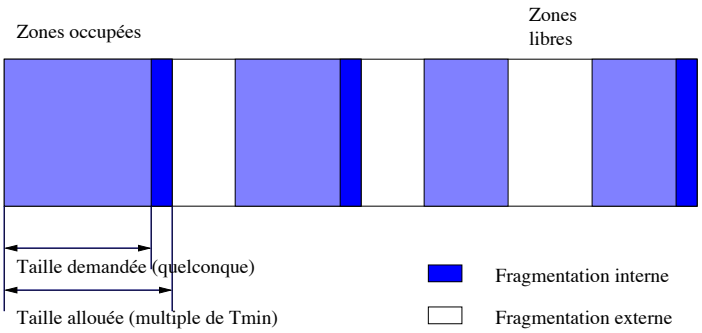
- Distinction entre zones libres et zones occupées
  ⇒ structure de données adaptée
  - Allocation : parcours de la structure de données pour trouver une zone libre
  - Libération : réintégration du bloc dans la structure de données

Allocation dynamique de mémoire
Fragmentation

- Fragmentation externe
  - Au fil des allocations/libérations, la mémoire est constituée d'un mélange de zones libres et occupées
  - Fusion de trous adjacents lors de la libération
  - La place prise par les zones libres peut être perdue si elles sont de trop petite taille
- Fragmentation interne
  - Place perdue quand l'allocateur mémoire alloue plus que demandé

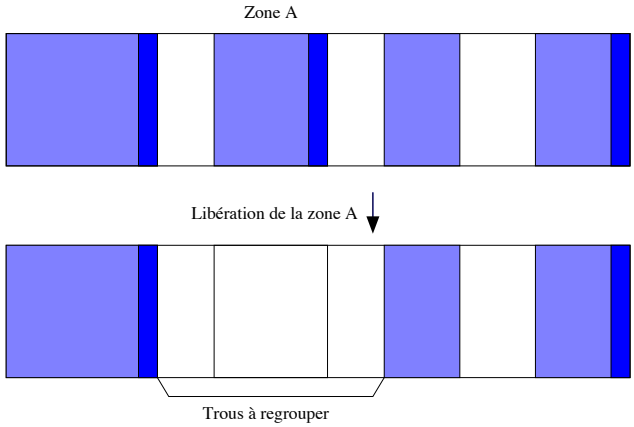
Allocation dynamique de mémoire Fragmentation

- Exemple avant fusion



Allocation dynamique de mémoire Fragmentation

- Exemple après fusion

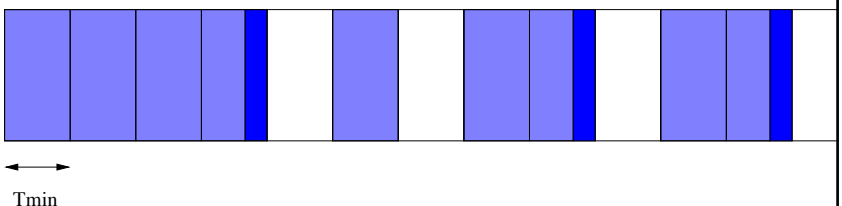


Allocation dynamique de mémoire
Classes d'algorithmes

- Bitmap : table de bits (1 bit par bloc)
- Sequential fits : structure de liste stockée dans les trous
- Indexed fits : autre structure de données (e.g. arbre) stockée dans les trous
- Buddy systems
- Politiques hybrides : dépendante de la taille de bloc demandée

Bitmap

- Allocation par multiple de bloc de taille fixée T_{min}
- Un bit par bloc (1 = bloc occupé, 0 = bloc libre)



Tmin

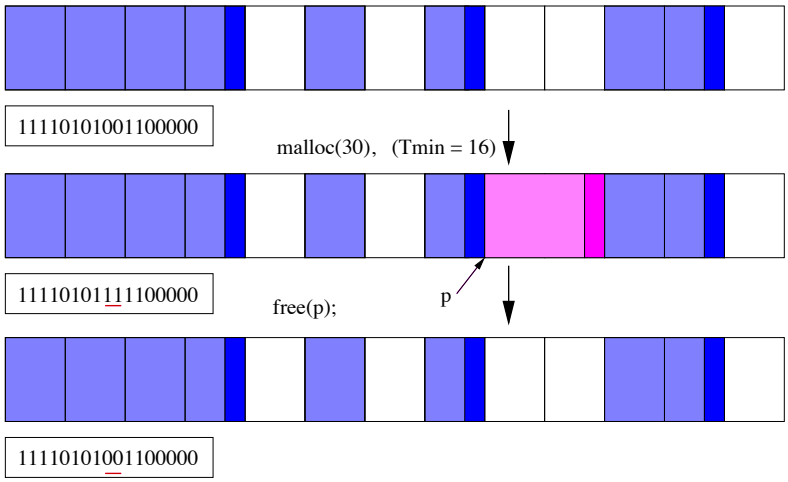
1111010110110000000

Bitmap

Bitmap

- Allocation
  - Arrondir la taille demandée au T_{min} supérieur ⇒ taille allouée = n*T_{min}
  - Recherche de n blocs consécutifs à 0, puis mise à 1
- Libération
  - Vérification dans la bitmap que la libération correspond bien à une zone allouée (bits à 1)
  - Mise des bits concernés à 0

Bitmap

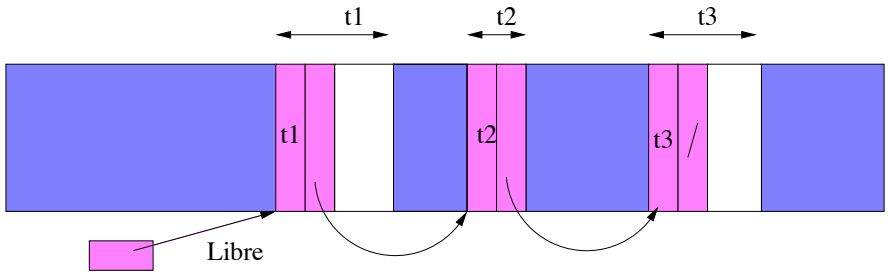


Bitmap

- Discussion
  - Adapté aux blocs de taille fixe (secteurs)
  - Instructions opérant sur les bits permet des réalisation efficaces

Sequential fits

- Chaînage des trous dans une liste
- Mémorisation de la structure de liste dans les trous

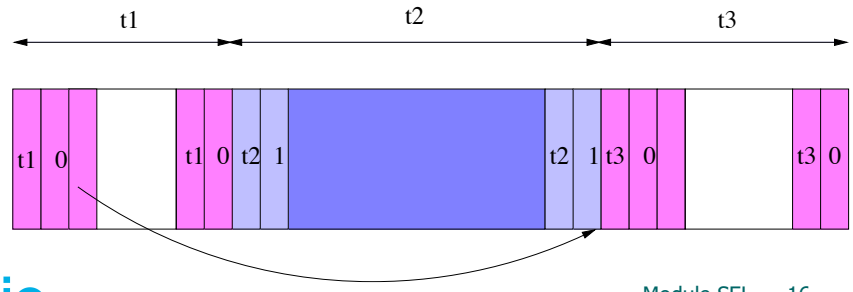


Sequential fits

- Allocation
  - Parcours de la liste des blocs libres
- Libération
  - Insertion dans liste des blocs libres
  - Fusion avec blocs adjacents si applicable pour limiter la fragmentation externe

Sequential fits

- Technique pour la fusion : boundary tags : pour tout bloc (libre ou occupé)
  - Entête (header) et prologue (footer) contenant :la taille du bloc l'état du bloc (libre - 0 - ou occupé - 1)



Sequential fits

- Stratégies courantes de recherche d'un bloc :
  - First fit : liste des trous triée par adresse, recherche du premier trou de la liste de taille ≥ à la taille demandée
  - Next fit : variation du first fit ou on gère la file circulairement en repartant lors de la recherche de la dernière zone allouée
  - Best fit : on recherche la plus petite zone convenable (paradoxalement, mauvaise utilisation de la mémoire due à une multiplicité de petits trous - résidus)

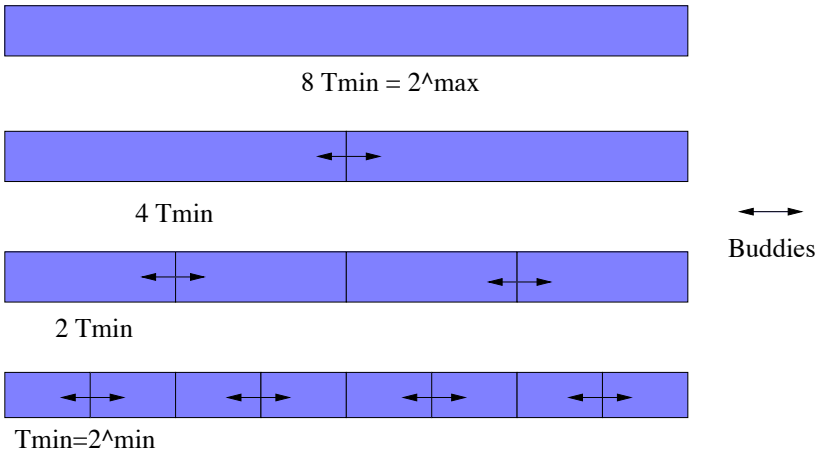
Amélioration des sequential fits

- Indexed fits : structures de données élaborée pour mémoriser les blocs libres
  - Arbre binaire équilibré permettant de trier les blocs par taille, arbre cartésien, etc …
  - Stockée dans les trous eux mêmes
- Segregated fits : structure de données et algorithme d'allocation différent par taille de bloc

Buddy systems

- Principe : on n'alloue que certaines tailles de blocs
  - Binary buddy : puissances de deux
  - Fibonacci buddy : taille membres d'une suite de Fibonacci
- Chaque bloc a son bloc compagnon (buddy) adjacent qui est le seul bloc avec qui il peut être fusionné en cas de libération
- Gros taux de fragmentation interne à cause des choix de tailles de blocs

Buddy systems



Buddy systems

- Liste de trous de taille 2^i
- Initialement, listes vides sauf 2^{max}
- Allocation

```
char *allouer(int T) {
        calcul de i tel que 2^(i-1) < T ≤ 2^i  a
        dr=trouver_trou(2^i);
        return (adr);
}
```


Buddy systems

```
char *trouver_trou (2^i ) {
   if (i > max) return -1;
   if (liste(i) vide) {
      ad=trouver_trou(2^(i+1));
       if (ad != -1) {
          diviser ce trou en 2 trous de taille 2^i
          placer ces 2 trous 2^i dans la liste(i)
      }
      else return -1;
   }
   adresse_trou = extraire_1er_trou_liste(i);
   retour adresse_trou;
}
```

Buddy systems

- Un petit exercice
- Initialement, la mémoire consiste en un bloc de 256K
  - Représenter l'état d'occupation de la mémoire après les allocations mémoire suivantes : A (5K), B(25K), C(35K), et D(20K) puis les libérations dans l'ordre A, D, C, et B

Allocation dynamique et mémoire virtuelle

- Tas (heap) = séquences d'adresses virtuelles
  - Tables des pages présente en mémoire
  - Pages physiques allouées en mémoire quand référencées
- Appels systèmes brk() et sbrk()
  - brk = adresse de fin du tas, brk() la retourne, srbk() étend le tas
  - Utilisé en interne par *malloc* (ne pas appeler directement)
  - Extension table des pages (pas d'allocation physique)

Glibc malloc – ptmalloc2

https://sensepost.com/blog/2017/painless-intro-to-the-linux-userland-heap/

- Arena : sous-tas pour autoriser l'allocation concurrentes aux threads
  - Limite l'attente due à la synchronisation
  - Nombre limité d'arenas
- Tailles arrondi au 16 bytes supérieur
- Allocateur hybride, listes de zones par tailles
  - Bins : listes doublement chaînées de blocs
    - 127 bins, 96 pour les petites zones, 21 pour les plus grosses
    - Regroupement immédiat avec trous adjacents

Glibc malloc – ptmalloc2

https://sensepost.com/blog/2017/painless-intro-to-the-linux-userland-heap/

- Listes de zones par tailles (suite)
  - Fastbins : file LIFO pour les touts petits blocs (<80) – candidats directs à la réutilisation, pas de merge
  - Bin Unsorted pour ré-utilisation immédiate avant de mettre dans le bon bin
  - Quand pas trouvé dans un bin, alloué en adresse croissante dans l'arena
- Bitmaps pour savoir quels bins sont vides/pleins
- La meilleure documentation est le code source
- Voir aussi https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/

Guide de survie de l'allocation dynamique (1/2)

- Ne pas l'utiliser quand on peut !
  - Sauf quand indispensable : taille et durée de vie d'une zone de mémoire inconnues a priori
- Taille connue (#define N 10)
  - *char tab[N];* et pas *char *tab=malloc(N);*
- Taille inconnue statiquement mais durée de vie connue (fonction). Possible même en C.
  - int f(int n) {
    int tab[n];
    for (int i=0;i<n;i++) tab[i]=i;
    }

Guide de survie de l'allocation dynamique (2/2)

- Ma mémoire est libérée en fin de processus ?
  - Oui, mais certains processus (serveurs) s'exécutent longtemps
- Quand j'alloue avec un *malloc* sur Linux, je ne consomme que de la mémoire virtuelle, ça ne coûte rien !
  - Si on en consomme trop, on swappe
- Mais madame, je fais toujours le *free* qui correspond au *malloc* (moi)
  - Même dans tous les cas d'erreur possibles ?
- Rien de va plus, votre programme fuit ? : valgrind

Questions / réponses

- Peut-on regrouper les trous d'une mémoire fragmentée, et si oui, sous quelle condition ?
- En supposant les informations de gestion accessibles, que se passe t'il en cas de :
  - Double libération ?
  - Utilisation après libération ?
  - Ecrasement des informations de gestion ?

Ramasse miettes

- Danger de la libération manuelle de mémoire (free)
  - Oubli de libération
  - Double libération
  - Utilisation d'une zone après libération
  - ⇒ Libération automatique de la mémoire (Ramasse-miettes, Garbage Collection)

Ramasse miettes

- Objet racine
  - Utile par définition (ex: pile)
- Objets utiles : accessibles directement ou indirectement à partir de l'objet racine via une chaîne de références

⇒ Nécessite de distinguer les références des données simples dans les objets

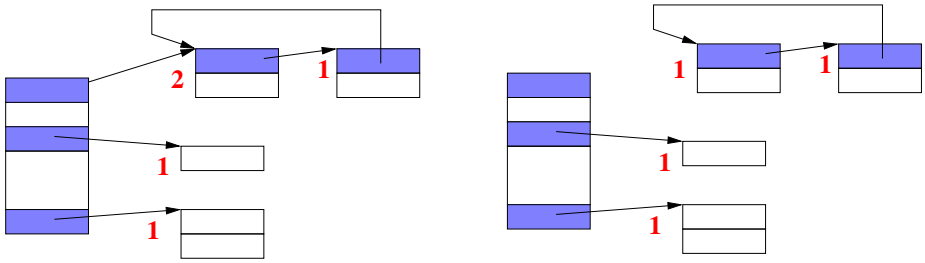
Ramasse miettes

- Classes de techniques
  - Comptage de références
  - Marquage et balayage
  - Ramasse-miettes générationnels, copiants, parallèles, incrémentaux, etc

Comptage de références

- Compteur de références par objet
- Ajout d'une référence : incrémentation du compteur
- Retrait d'une référence : décrémentation du compteur
- Destruction de l'objet quand son compteur de références atteint 0
- Utilisé dans les SGF pour la destruction des fichiers (liens physiques)

Comptage de références



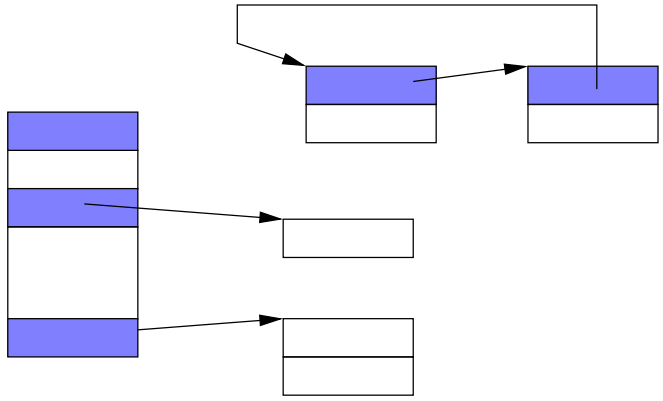
Comptage de références

- Bénéfices
  - Rapide
  - Naturellement entrelacé avec l'exécution de l'application
- Inconvénients
  - Ne libère par les structures cycliques

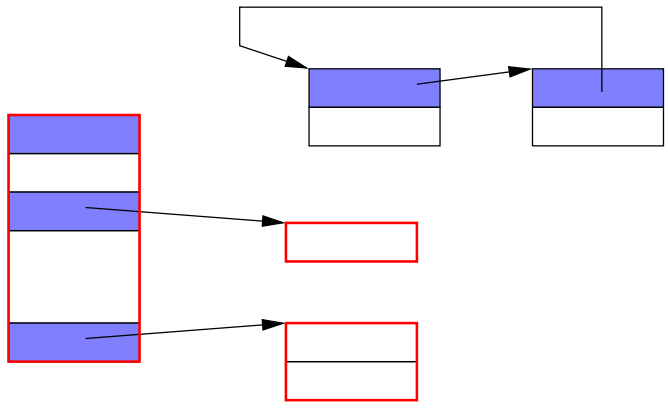
Marquage et balayage (mark and sweep)

- Marquage
  - Marquage des objets racines
  - Marquage de tout objet non marqué référencé par un objet marqué
  - (Parcours du graphe des références)
- Balayage : libération de la mémoire de tout objet non marqué

Marquage et balayage (mark and sweep)



Marquage et balayage (mark and sweep)



Marquage et balayage (mark and sweep)

- Avantages
  - Détruit tous les objets inutiles
- Inconvénients
  - Pause pouvant être longue à chaque exécution

  ⇒ exécution en parallèle avec l'application (parallèle, incrémental)

  ⇒ on peut en profiter pour regrouper les blocs et éviter la fragmentation (et augmenter la localité !)