Gestion de la Mémoire Dynamique

Partie 1 : Tour d'Horizon des Allocateurs Classiques

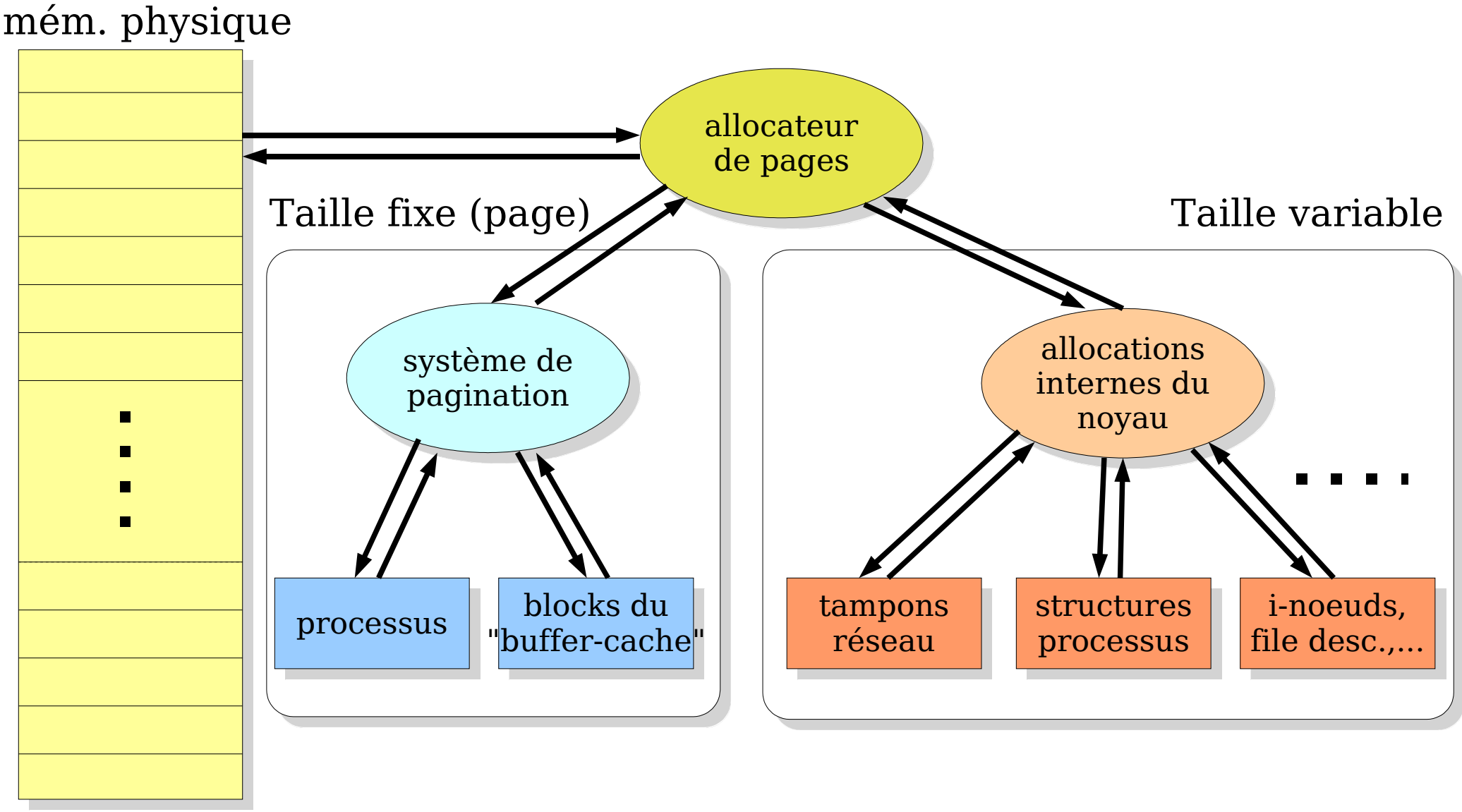
Rappel : la Gestion Mémoire en Mode Utilisateur

- Allocations dynamiques en mode utilisateur
  - Fonctions de bibliothèque
    - Allocations persistantes
      - `malloc` : allocation persistante
      - `calloc` : allocation persistante avec init à 0
      - `free` : libération
      - `realloc` : redimensionne (ou libère)
    - Allocation temporaire (sur la pile)
      - `alloca` : libération automatique (non standard)
    - Allocation bas niveau
      - `sbrk` : augmente la taille du segment de données
        - `sbrk(0)` retourne la valeur courante
  - Appel système
    - `brk(void *)` : fixe l'adresse de la fin du seg. données.
      - Tout le reste est construit à partir de ça par la libc !

Mémoire Dynamique en Mode Noyau

- Pour quel usage ?
  - Affectation de **pages** à différents sous-systèmes
    - E/S block (cache de pages), E/S matériel (DMA)
    - Affectation de pages aux processus
      - Réponse aux appels brk
      - La page est la granularité minimale d'allocation
  - Allocation internes du système
    - Noyau
      - Pourquoi faire ?
        - besoins non connus à la compilation
        - besoins non connus au moment du bootstrap
      - Allocations dynamiques courantes
    - Modules :
      - Allocation des segments initiaux (infos dans le code objet)
      - Allocations dynamiques courantes (kmalloc/kfree)

Allocateurs Classiques dans les Noyaux Unix



Critères d'évaluation et contraintes

- Minimiser le gaspillage
  - métrique : facteur d'utilisation
    - Quantité réellement utilisée pour satisfaires requêtes
    - Idéalement 100 %, en pratique 50% est acceptable ...
- Rapidité
  - nb instructions en moyenne **et** dans le pire cas
  - optimisation de la localité (maximise cache-hits)
- Simplicité de l'API
  - malloc/free est un bon exemple
    - Avantage : inutile de se rappeller la quantité allouée
    - Inconvénient : pas de libération partielle
- Respect de contraintes d'alignement
  - caches, contraintes CPU (8octets sur CPUs alpha64)

Exemples d' Allocateurs - 1 : Table de Ressource

- Ressource = zone de mémoire libre
  - décrite par un couple <@base,taille>
  - Au début : unique ressource = zone entière
- Principe de fonctionnement
  - entrées triées dans l'ordre croissant des @base
  - fusion des zones libres contigues lors de la libération
  - Trois stratégies d'allocation
    - First Fit : la première qui est assez grande
      - Rapide, mais génère beaucoup de fragmentation
    - Best Fit : la plus petite qui soit assez grande
      - Le plus lent, mais minimise la taille des fragments
      - Génère de nombreux fragments inutilisables
    - Worst Fit : à moins de trouver exactement, choisi le plus grand fragment disponible

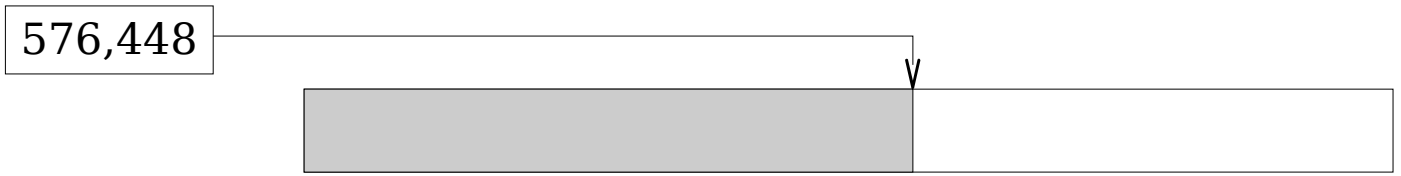
Avantages de l'Allocateur TR

- Facile à implémenter
- Assez générique
  - Ex: allocation de pages contigües
  - Espace disque (secteurs dans une partition)
  - Emplois du temps (allocation de 1/4h pour les séances)
- Supporte la libération partielle
  - fusion éventuelle de la zone libérée avec précédente ou suivante dans la table
- Réutilisation de la mémoire libérée pour des allocations de tailles différentes

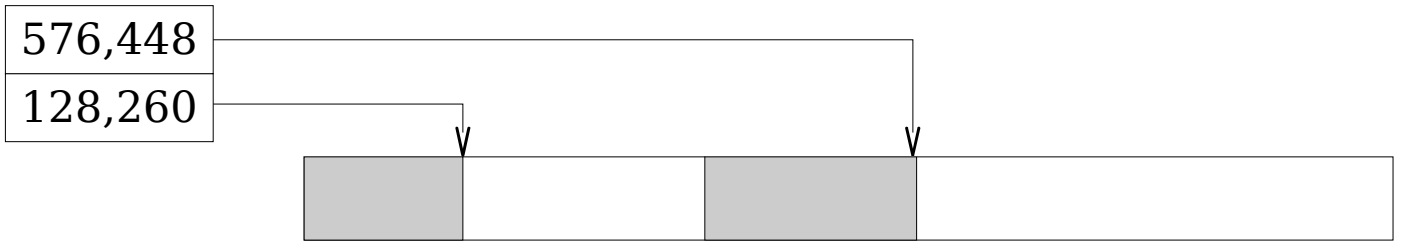
Exemple d'Utilisation de l'Allocateur TR

1. Au début

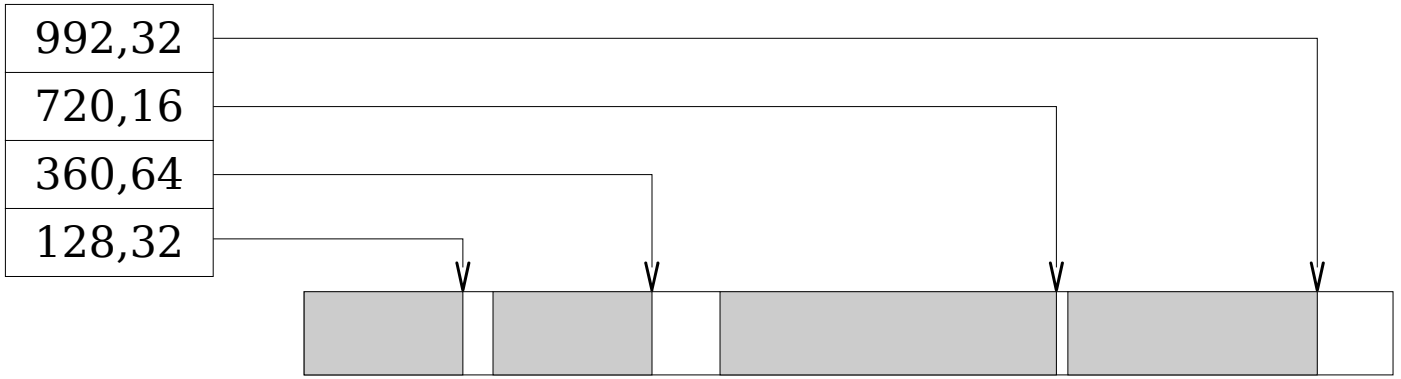
2. malloc(576)



3. free(128,260)



N. ...



Inconvénients de l'Allocateur TR

- A la longue, la table devient très fragmentée
- La taille de la table augmente avec la fragmentation
  - Pb si table de taille fixe
  - Besoin d'un allocateur si taille variable (!)
- Pour la fusion, les entrées doivent être triées
  - En particulier si le tri doit se faire "sur place"
    - Cas où la table est stockée sous forme de vecteur
- L'allocateur doit parcourir la table
  - Complexité O(nb entrées) : augmente avec la fragmentation
    - Pire cas ??
- Pas prévu pour rendre la mémoire du "pool" (au syst. de pagination)
  - La mémoire allouée en cas de "burst" ne peut être rendue

Exemples d'Allocateurs – 2 : Puiss.-de-2 Simple

- Fréquent dans les allocateurs malloc/free de la libc
- Basé sur un ensemble de "free-lists"
  - Chaque liste stocke des blocs libres d'une taille donnée
  - Les **tailles** sont des **multiples d'une puissance 2**
    - 7 listes si 16 < taille <1024 : 16, 32, 64, 128, 256, 512, 1024
- Chaque bloc comporte un en-tête
  - Réduit d'autant la taille du bloc alloué (à prévoir)
  - Bloc alloué :
    - Contient l'@ de la liste où le bloc devra être restitué
      - variante : contient la taille
  - Bloc libre :
    - Contient l'@ du bloc libre suivant dans la liste

Exemples d'Allocateurs – 2 : Puiss.-de-2 Simple

- Allocation
  - Calculer le plus petit bloc possible
    - En tenant compte de l'en-tête
  - Retirer le premier dans la liste correspondant à cette taille
  - Alternatives si aucun bloc n'est dispo pour cette taille
    - Prendre le premier bloc venu de taille supérieure
    - Réclamer de la mémoire au syst. de pagination pour peupler la liste avec des "blocs frais"
    - Bloquer (endormir) l'appelant jusqu'à ce qu'un bloc soit libéré
- Libération
  - Remonter jusqu'à l'en-tête pour trouver la liste
  - Le bloc est retourné en tête de liste
    - LIFO est plus efficace pour les caches

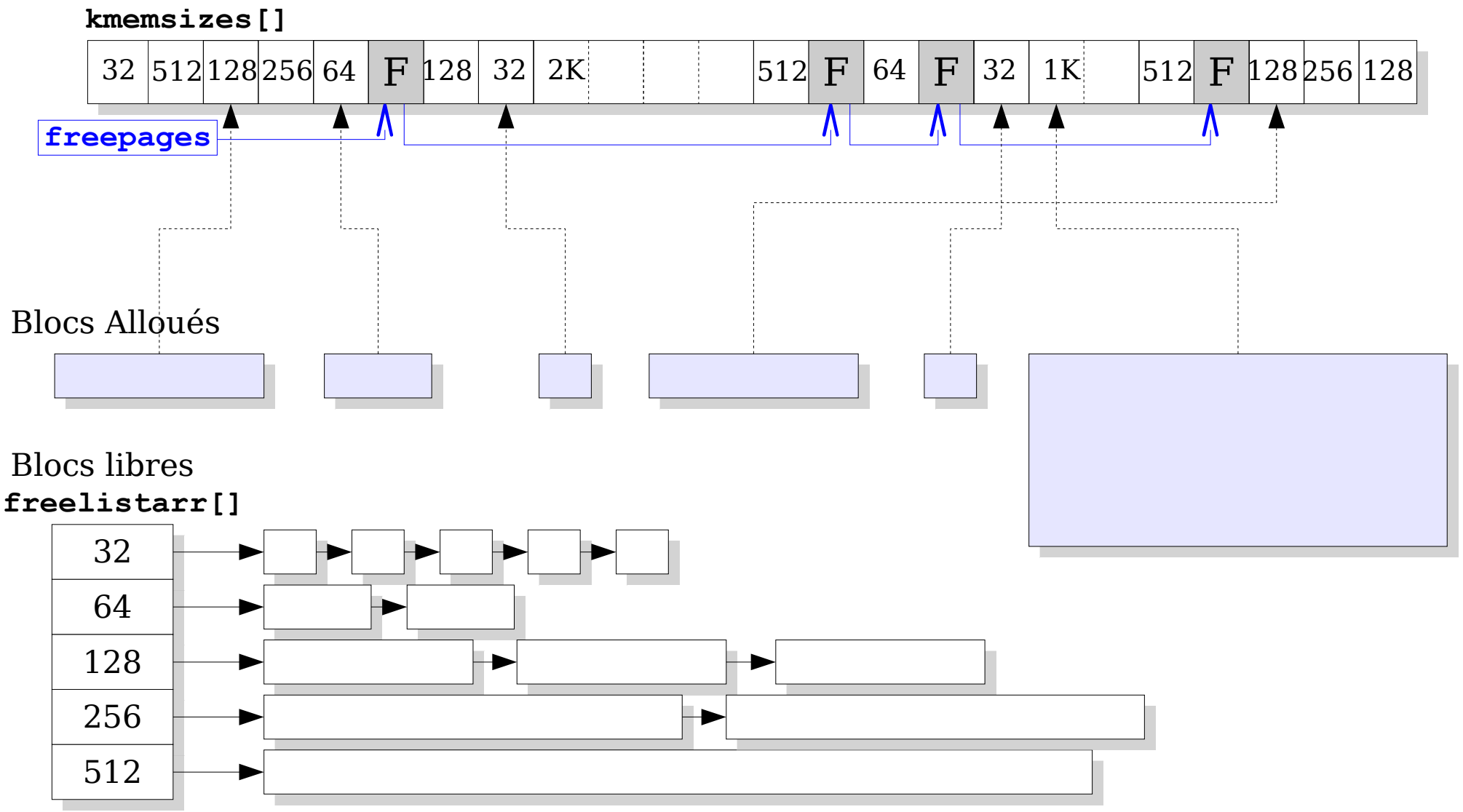
Analyse de l'Allocateur P2-S

- Avantages
  - Evite une couteuse recherche linéaire
  - Performance du pire cas bornée
    - Quand le bloc demandé est dispo
  - Similaire à free()
    - Inutile de préciser la taille du bloc libéré
- Inconvénients
  - Fragmentation interne importante
    - En particulier pour allocation de blocs de taille puiss. de 2
  - Fusion de blocs impossible pour satisfaire demandes de taille plus importante
    - Une forme de fragmentation externe

Exemples d'Allocateurs – 3 : McKusick-Karels (utilisé dans 4.4BSD, Digital Unix)

- Version améliorée de l'allocateur "puissance-de-2"
  - Requiert un pool formé de pages contigües
    - Facile pour le noyau
  - Dans une page donnée, toutes les allocations ont même taille
  - Utilise un tableau supplémentaire : kmemsizes[]
    - Chaque page dans l'un des 3 états suivants
      - Libre : kmemsizes[i] pointe sur la page libre suivante
      - Divisée en morceaux : kmemsizes[i] contient la taille des morceaux
      - Fait partie d'un morceau (plus gros qu'une page) : kmemsizes[i_0] contient la taille
  - **Plus besoin d'en-tête**
    - La taille peut être déduite du N° page grâce à kmemsizes[]
      - N° page = adresse décalée n bits à droite

Schéma de Fonctionnement de l'Allocateur McK-K



Analyse de l'Allocateur McK-K

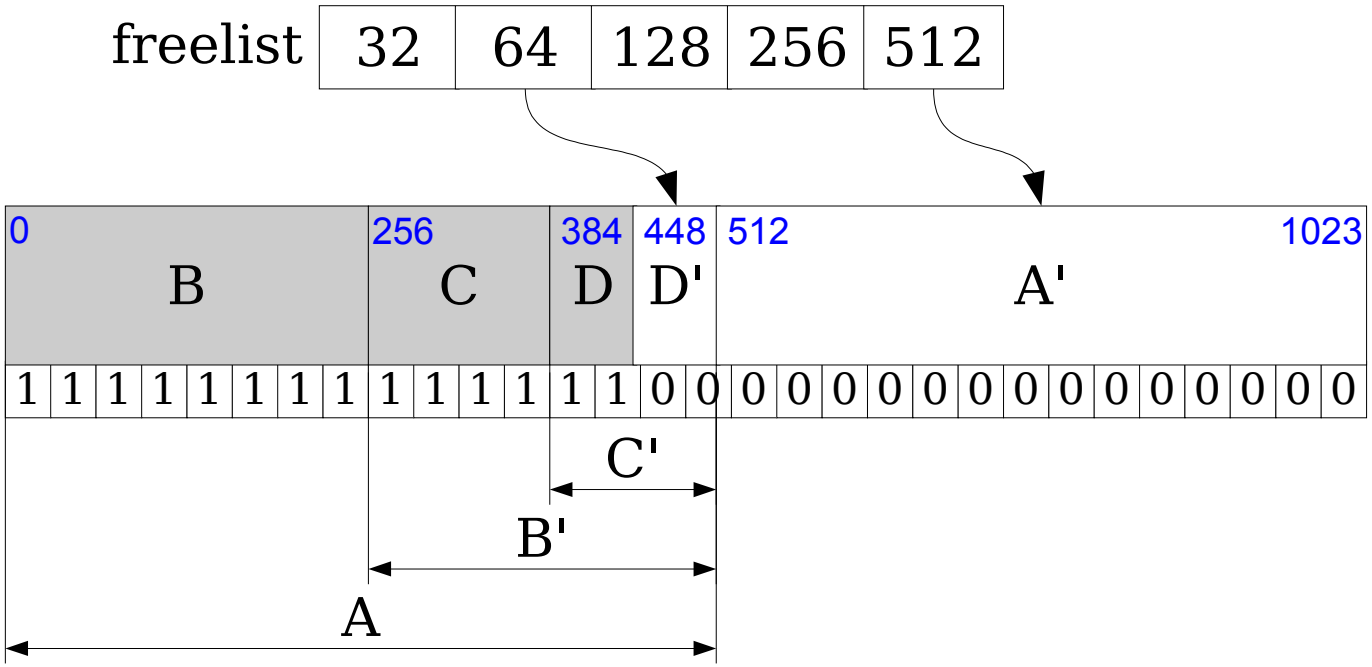
- Avantages
  - malloc/free implémentées sous forme de macros C
    - Optimisation importante si taille alloc connue à la compilation
  - Plus de fragmentation pour des tailles multiples de 2^n
  - Moins de gaspillage
- Inconvénients
  - Pas possible de faire passer un morceau d'une liste à une autre
    - Allocateur sensible aux phénomènes du "burst"
      - Quand momentanément une grande qté de mémoire est utilisée pour allouer des blocs d'une taille donnée
  - Imossible de rendre la mémoire au système de pagination
    - Allocation contigüe des pages (difficile de faire un trou)
    - Recherche coûteuse des pages libérables

Exemple d'Allocateurs – 4 : Le "Buddy Allocator"

- Buddy = binôme
  - Combine fusion des zones libres avec allocateur 2^n
    - Crée des petits buffers par découpages successifs en 2
    - Exemple : on veut 256, on n'a qu'un bloc de 1024
      - Découpage de 1024 en 2x512 (blocs binômes)
        - bloc de 1024 retiré de la liste 1024
        - 2 blocs de 512 placés dans la liste des 512
      - Découpage de 512 en 2x256 (blocs binômes)
        - (un) bloc de 512 retiré de la liste 512
        - 1 bloc de 256 placé dans la liste 256
        - 1 bloc de 256 retourné
    - Fusion
      - Un bitmap indique l'utilisation des tronçons de +petite taille
      - Binôme dans liste des blocs libres ?
        - (sous-)bitmap des tronçons du binôme = 0

Schéma de Fonctionnement de l'Allocateur BA

- Exemple :
  - Tailles supportées : de 32 à 512 octets
    - 32o : taille du tronçon du plus petite taille
    - bitmap = 1024/32 = 32 bits
  - Taille du pool : 1024



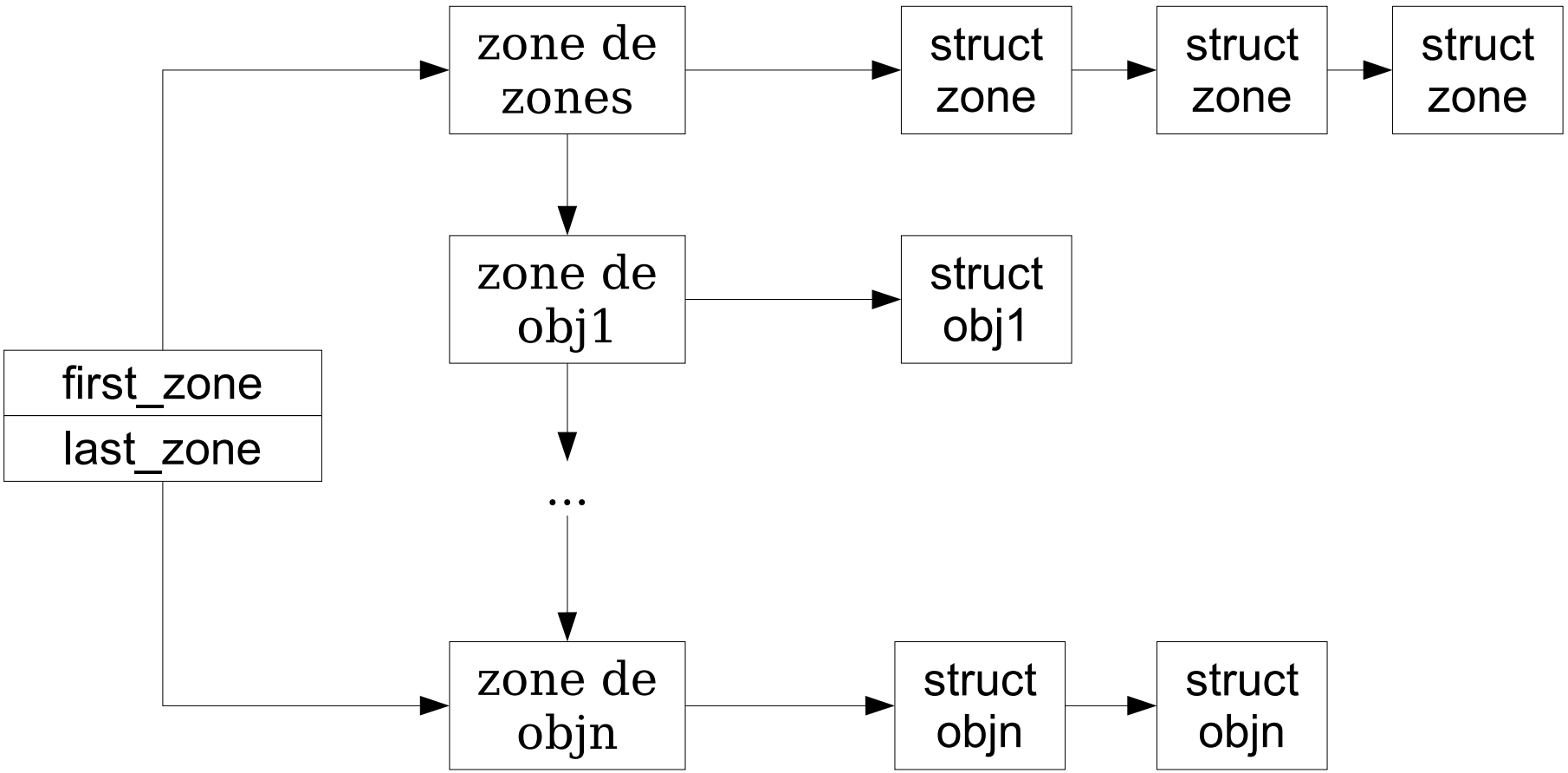
Analyse de l'Allocateur BA

- Avantages
  - Meilleure utilisation de la mémoire que 2^n
    - Utilise toujours la puiss de 2 immédiatement supérieure
    - Mais toujours les défauts de l'arrondi à 2^n
      - fragmentation interne potentiellement importante
  - Capable d'agrandir ou réduire le pool
    - Ajout/retrait de page facile
- Inconvénients
  - Performances
    - Multiples fusions possibles à chaque libération
    - Cas pire assez dramatique
      - pool totalement inutilisé :
        - Allocation 32, libération 32, allocation 32, libération 32, ...
      - Défault corrigé dans SVR4 (lazy-coalescing)

Exemples d'Allocateurs : 5 – Allocateur de Zone

- Une zone pour chaque type d'objet dynamique
  - Pool d'objets libre dans chaque zone
  - Objets de type différent ont des zones distinctes
    - Même s'ils ont la même taille !
  - Chaque zone est peuplée à l'aide de l'allocateur de pages
  - Une zone est de taille bornée
    - Max spécifié lors de la création de la zone
      - Chaque zone a son propre max
    - Echec des allocation lorsque le max est atteint
- Un ramasse-miette travail en arrière plan
  - Compteur d'utilisation associé à chaque page d'une zone
    - Restitution des pages entièrement inutilisées

Scéma de Fonctionnement de l'Allocateur ZA



Analyse de l'Allocateur ZA

- Allocation et libération très rapide
- L'utilisation du rammasse-miette évite les instabilités
  - Cas limites pathogènes du même type que dans le BA
- Interface simple
  - Création/Destruction d'une zone
  - Allocation/Libération d'un objet d'une zone donnée
- Allocation initiale de zone de zones ?
  - Résolu par allocation statique lors du boot
- Pas de problème de fragmentation interne
- Fragmentation externe réduite
  - Restitution des pages inutilisées
- Aspect critique(able?) : le ramasse-miette
  - Exécution couteuse (parcours linéaire des zones)

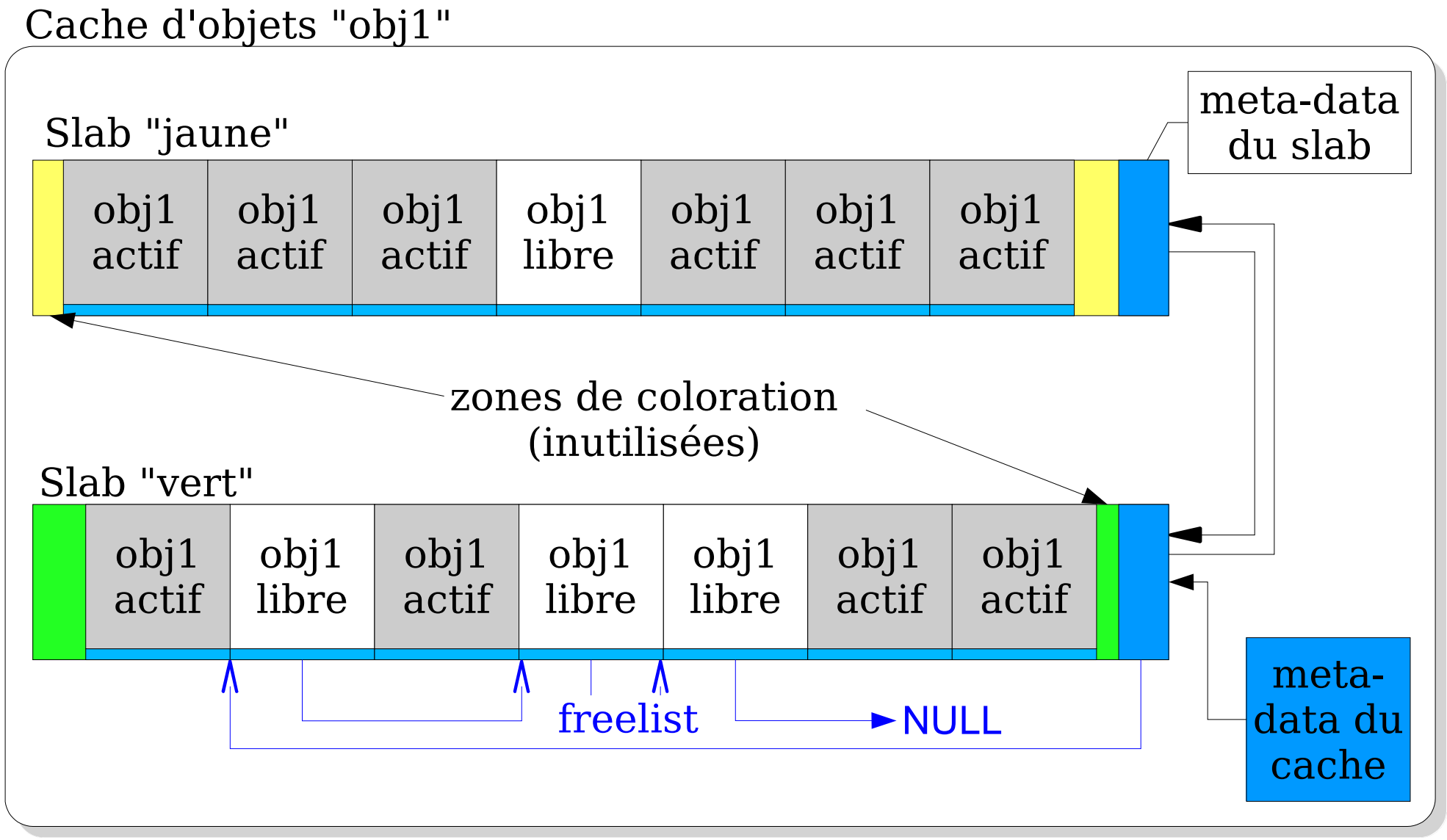
Exemples d'Allocateurs : 6 – Le "Slab-Allocator"

- Même idée générale que le ZA
  - Conçu pour être extrêment performant
    - Prise en compte de nombreux aspects ignorés par les autres
      - Exploitation propriétés des caches
      - Optimize la réutilisation (déconstruction intelligente)
      - Gestion différente pour petits et gros objets
        - Petits objets : meta-data dans chaque page utilisée
        - Gros objets (multi-pages) : meta-data en dehors
- S'appuie sur l'analyse du cycle de vie d'un objet
  - Allocation
  - Initialisation
  - Utilisation
  - Déconstrution
  - Libération

Exemples d'Optimisations du SA

- Optimisation des initialisations/déconstructions
  - Un objet libéré est souvent "propre à la consommation"
    - Ex : Compteur de référence avec valeur initiale
    - Inutile de l'effacer lors de la prochaine allocation !
  - Déconstrution
    - Préparer l'objet pour une future initialisation rapide
- Optimisation pour les caches
  - Eviter de se promener partout en mémoire
    - Rapprocher les données utiles à l'allocation
    - Limiter le nombre d'accès aléatoires à la mémoire
  - Un cache fonctionne de façon optimale quand les objets ne sont pas tous alignés de la même façon
    - Principe de coloration des zones
      - jouer sur la marge due à la fragmentation pour aligner ≠ment

Schéma d'Organisation du SA (cas de Petits Objets)



Initialisation et Déconstruction du SA

- Idée sous-jacente
  - objet réutilisé : ni déconstruit, ni initialisé
    - Il doit être restitué **dans son état d'utilisation initial**
    - Permet d'économiser le temps d'opérations souvent symétriques
  - Oui quand au début, ou quand le cache "grossit" ?
    - Les fonctions d'initialisation et déconstruction sont fournies lors de la **création du cache**
    - cachep = kmem_cache_create(name,size,align,**ctor**,**dtor**)
    - ctor/dtor invoquée lors de l'allocation restitution de nouvelles pages dans le cache

Analyse du SA

- Avantages
  - Ultra-performant
    - Allocation rapide
    - Utilisation efficace des caches matériels
    - Empreinte mémoire (footprint) réduite
  - Gaspille peu de mémoire
  - Ramasse-miette moins couteux que ZA
    - compteur d'utilisation évite la recherche linéaire
  - Pire cas proportionnel au nombre de caches
    - pas au nb de slabs
- Inconvénient
  - Couteux quand un cache est peu utilisé
    - Idée : créer des caches pour allouer des buffers généraux 2^n

Comparaison de performances

	BA paresseux (SVR4)	McK-K (BSD4.4)	SA (Solaris)
temps moyen d'alloc/libération (ms)	9,4	4,1	3,8
Fragmentation totale (gaspillage)	46,00%	45,00%	14,00%
Kenbus benchmark (nb scripts/sec)	199	205	233

(Résultats tirés de "Unix Internals" (Vahalia))