LIBK

MI074 - 3

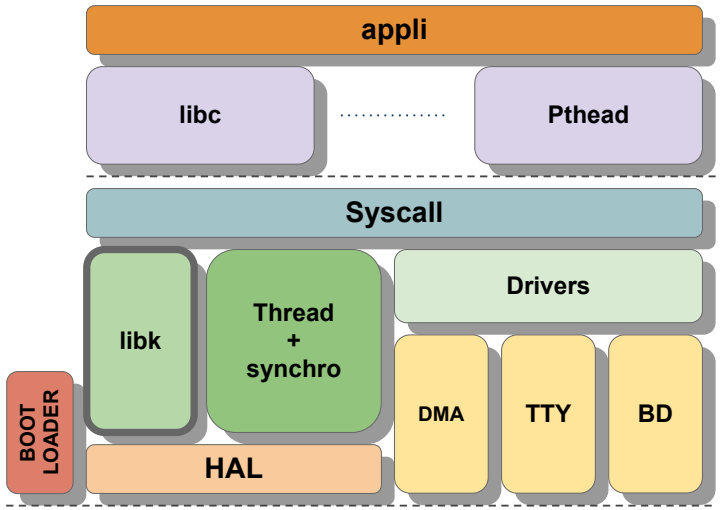
libk

- Fonctions auxiliaires dont le système pourrait avoir besoin.
  C'est un sous-ensemble de la libc, sauf qu'il n'y a pas de syscall.
  - int kprintf(const char *restrict format, ...);
    int kgetc(void);
    char * kgets(char * restrict str, int size);
    - Affichage formaté de messages pour le système
      Lecture d'un caractère ou d'une ligne (jusqu'au *return*)
  - void * kbzero (void *s, unsigned n);
    void * kmemmove(void *s, const void *d, unsigned n);
    char * kstpcpy(char *s, const char *d);
    int kstrcmp(const char *s1, const char *s2);
    int kstrlen(const char *s);
    long kstrtol(const char *restrict str, char **restrict endptr, int base);
    int krand(void);
    void ksrand(unsigned seed);
    - Conversion chaine -> entier
      Déplacement de zone de mémoire
      Fonctions de manipulations des chaines
      Générateur aléatoire
- Gestion de la mémoire dynamique
  - int kmalloc_init (void *start, void *limit);
    void* kmalloc(unsigned size);
    void kfree(void *ptr);
    void kmalloc_status(void);
    - Allocation dynamique dans le tas du système
- Gestion de liste doublement chainées

Matériel utilisé

- Le système se réserve l'usage du TTY0
- *spinlock*
  Dès lors qu'il y a plusieurs threads en // et des ressources partagées, il faut un moyen de garantir l'accès atomique

Architecture de l'OS final



kprintf

extern int kprintf(
    // chaine format à produire
    const char *restrict format,
    // liste variable des arguments
    ...);

restrict indique que le buffer ne sera accéder qu'avec le pointer **format**
const indique que **format** est en lecture seul

kprintf renvoie le nombre de caratères imprimés
La taille des message imprimés est borné par une constante MAX_KPRINTF

http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man3/stdarg.3.html

```
#include <stdarg.h>
void
foo (char *fmt, ...)
{
  va_list ap;
  int     d;
  char    c, *p, *s;
  va_start  (ap, fmt);
  while     (*fmt)
    switch  (*fmt ++) {
      case 's': /* chaîne */
        s = va_arg (ap, char *);
        printf ("chaine %s\n", s);
        break;
      case 'd':  /* entier */
        d = va_arg (ap, int);
        printf ("int %d\n", d);
        break;
      case 'c':    /* caractère */
        c = va_arg (ap, char);
        printf ("char %c\n", c);
        break;
    }
  va_end (ap);
}
```

Gestion mémoire

Un programme peut créer des structures dans 3 types d'espace:

1. Dans des variables globales dont les adresses sont connues dès la phase de compilation-édition des liens.
2. Dans des variables locales dans la pile. C'est un espace relatif au pointeur de pile qui n'est utilisable que pendant la durée de vie de la fonction qui déclare l'espace.
3. Dans le tas (heap). C'est un espace global géré dynamiquement dans lequel on peut réserver des zones contiguës par malloc et que l'on doit explicitement libérer par free.

Solution naïve

Si la mémoire allouée n'est jamais libérée alors on peut utiliser une solution naïve mais efficace utilisant un pointeur de limite.

On suppose disposer :
- d'un espace contiguë de grande taille.
- d'un pointeur **top** initialisé sur la 1ère case de cet espace.

A chaque nouveau malloc :
- on se contente de rendre la valeur du pointeur **top**.
- et on repousse **top** de la taille de la zone demandée.

Fonctions de la mémoire dynamique

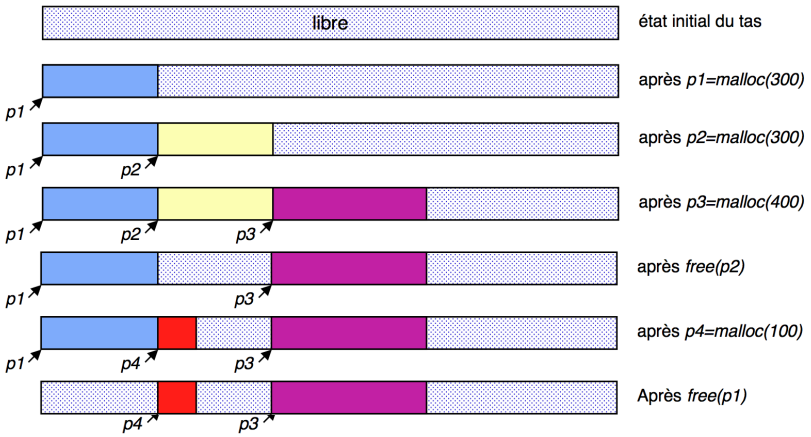
void *malloc(size_t size)

renvoie un pointeur vers un bloc de mémoire 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.

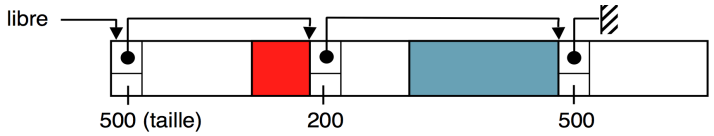
Gestion malloc-free



- Pour libérer et allouer dans les espaces libérés (p4) il faut se souvenir de la taille des zones allouées.
- On voit apparaître un phénomène de fragmentation qui entraine de perte de mémoire

Structuration et recherche dans le tas

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



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

1. **First Fit :** La première zone libre assez grande pour satisfaire la demande.

   Avantage : rapidité de la réponse.
   Inconvénient : risque de mauvaise utilisation de l'espace libre (perte de grands segments).

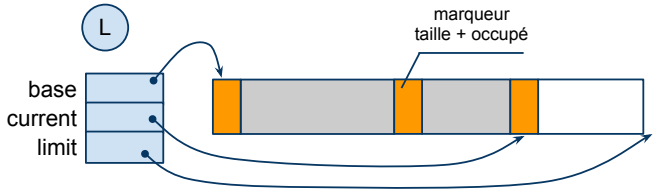
   Next Fit : Optimisation la recherche commence à partir de la position courante du pointeur de parcours et non à la première zone libre.

   Avantage : évite l'accumulation de petites zones libres en début de liste.

2. **Best Fit :** La zone dont la taille est plus proche de la demande.

   Avantage : meilleur ajustement.
   Inconvénients : plus lent; risque de créer des zones de faible taille (émiettement)

Implémentation des pointeurs de zones

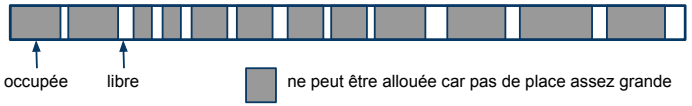


- Les pointeurs de zones sont remplaçables par des marqueurs qui codent sur un entier :
  - la taille de la zone sur 31 bits
  - son état d'occupation sur 1 bit
- Connaissant le pointeur de début (base ou current), on peut parcourir les zones à la recherche d'espace libre.
- Lors des libérations on peut fusionner les zones contiguës (dans les adresses croissantes)
- Lors de l'allocation, s'il n'y a plus d'espace, on peut parcourir le tas à partir du début pour fusionner les zones libres

Problème de fragmentation

A l'usage lorsque l'on demande beaucoup d'allocation et libération, on fabrique des segments libres de taille trop petites pour être utilisables.

La **fragmentation externe** est l'espace **entre zones allouées**

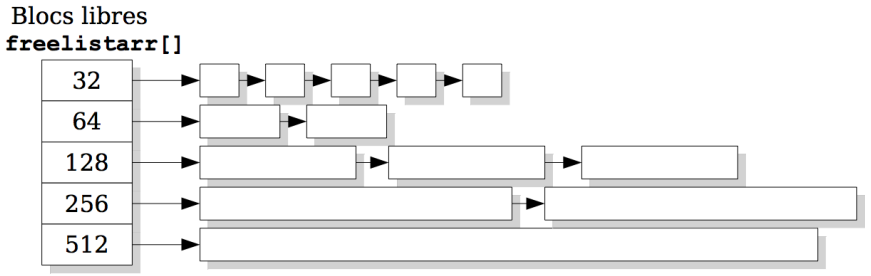


Pour éviter d'avoir des zones libres inutilisables entre les zones allouées on partitionne le tas en blocs atomiques.
Les zones allouées sont des multiples entiers de blocs (de parties).

La **fragmentation interne** est l'espace inutilisé **dans les blocs**

En outre:
La taille des blocs est un nombre entiers de ligne de cache pour réduire le taux de miss des caches et pour éviter des faux partages de lignes entre cores.

Optimisations



- Pour réduire la fragmentation externe, on peut créer plusieurs tas où chaque tas gère un intervalle de taille.
- Lorsque que le tas est très utilisé avec un grand nombre d'allocation-libération. on peut créer des pools de buffers de taille standardisée. L'allocation initiale est faite dans le tas, la libération attache la zone libérée à une liste de buffer libres (1 liste par taille).

Les listes chainées

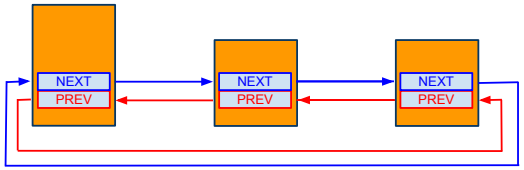
La plupart des structures du données du système utilise des listes chainées. Il faut disposer d'une API pour les manipuler de façon homogène et lisible sans ajouter de surcoût en mémoire et en temps vis à vis d'une solution traditionnelle.

Il faut :

- créer des listes
- insérer des éléments
- détruire des éléments
- parcourir les listes

Gestion des listes chainées API

```
list_root_init(list_t *root)

list_foreach_forward(root, iter)
list_foreach_backward(root, iter)
list_foreach(root, iter)

list_first(root, type, member)
list_last(root, type, member)
list_element(current, type, member)

list_empty(root)
list_isfirst(root, current)
list_islast(root, current)

list_add_first(root, entry)
list_add_last(root, entry)

list_add_next(current, entry)
list_add_pred(current, entry)

list_unlink(list_t *entry)
list_replace(list_t *current, struct list_t *new)
```

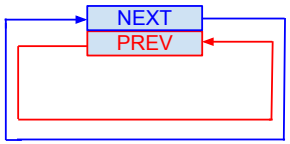


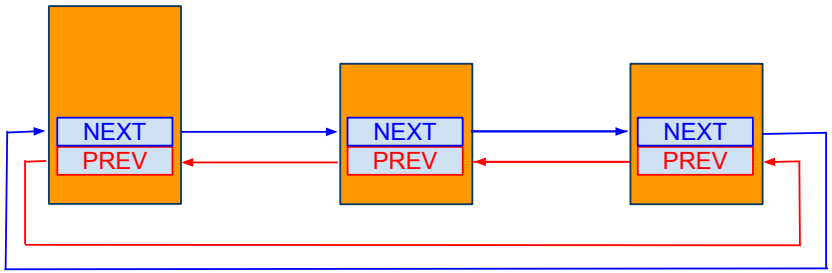
Gestion des listes chainées

Le système utilise des listes chainées partout.
Le noyau exige un mécanisme générique et efficace.

- Déclaration
- Itérateur
- Fonctions d'accès
- Exemple

struct list_entry





Gestion des listes chainées : exemple

```
#include <klist.h>
#include <stdio.h>

// Definition du type famille
struct famille_s
{
     char *nom;
    list_t root;
};


// Definition du type personne
struct personne_s
{
    char *prenom;
    list_t list;
};
```

```
int main()
{
    struct famille_s dupont;
    struct personne_s jean_claude;
    struct personne_s monique;

    dupont.nom = "DUPONT";
    jean_claude.prenom = "Jean-Claude";
    monique.prenom = "Monique";

    // Initialisation de la racine de list
    list_root_init( &dupont.root);

     // Ajout d'un personne
     list_add( &dupont.root, &monique.list);

     // Ajout d'une autre personne
    list_add( &dupont.root, &jean_claude.list);

    printf("La famille %s :\n", dupont.nom);

    list_foreach( &dupont.root, iter) {
        struct personne_s *ptr = \
            list_element( iter, struct personne_s, list);
        printf( " - %s\n", ptr->prenom);
    }

    return 0;
}
```