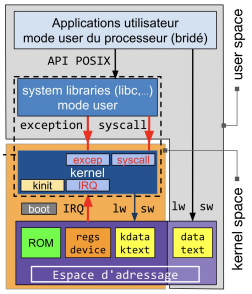
Allocateurs de mémoire

LU3IN031 Architecture des ordinateurs - 2
Matériel et Logiciel
B6

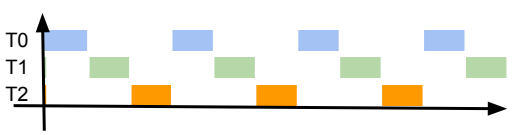
franck.wajsburt@lip6.fr
V4

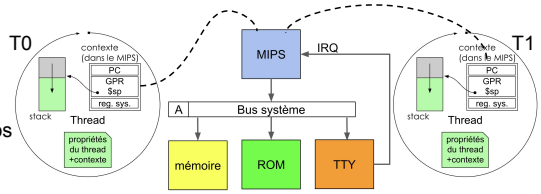
Ce que nous avons vu

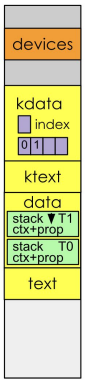
- Nous avons déjà vu les 3 gestionnaires de services du noyau
  - Le gestionnaire d'appel système permettant aux applications de demander un service ou une ressource
  - Le gestionnaire d'interruptions permettant aux contrôleurs de périphériques de voler des cycles au applications en cours d'exécution pour gérer un événement.
  - Le gestionnaire d'exception pour les erreurs du programme.



- Nous avons vu l'exécution d'un processus en temps partagé
  - Les processus sont des conteneurs de ressources pour l'exécution des applications
    - un programme, c.-à-d. le code à exécuter ;
    - un espace d'adressage, c.-à-d. des mémoires pour le code et les données ;
    - des ressources, c.-à-d. des fichiers, des zones d'échanges, des processeurs ;
    - des threads, c.-à-d. un contexte et une pile ;
  - Les threads utilisent le processeur à tour de rôle. Quand, ils ont le processeur, ils le rendent quand ils sont bloqués sur une ressource ou quand leur temps est fini.







Objectifs de la séance

Les données manipulées par l'application sont placées dans les variables globales et dans les piles de threads. Si l'application a besoin de place pour des données partagées par les fonctions ou par les threads, elles ne doit pas utiliser le pile, il faut un espace pour allouer dynamiquement des segments de mémoire : c'est heap (tas)

- Comment créer des objets de tailles quelconque pour l'application et pour le noyau ?
- Est-ce qu'un seul type d'allocateur peut fonctionner pour tous les usage ?
- Quelles conséquences, l'existence des allocateurs a sur la gestion des threads ?

Plan

1. Les besoins en mémoire pour le noyau et les applications
2. API de gestion de listes doublement chaînées avec itérateur
3. Allocateurs de piles utilisateur
4. Allocateur first fit pour l'application
5. Allocateur slab pour les objets du noyau
6. Conséquence sur la gestion des threads

Besoins en mémoire

Les données de l'application utilisateur

Variables globales

L'application dispose des variables globales définies dans le code du programme, allouées lors du chargement du programme en mémoire.

➡ Elles sont dans le segment data, nombre et taille fixes

Variables locales

Chaque thread dispose d'une pile pour la sauvegarde des contextes de fonctions dans lesquels se trouvent aussi les variables locales (sauf les variables static). Hormis le premier thread (**`main`**), tous les threads sont créés dynamiquement, à la demande de l'application.

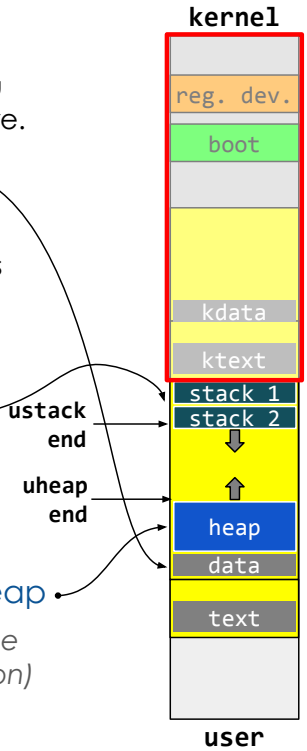
➡ Il faut donc un allocateur pour les piles de threads

Variables dynamiques

Chaque thread peut demander dynamiquement des segments de mémoire avec une API « **`malloc()`** » / « **`free()`** »

➡ Il faut un allocateur d'objets dans un segment nommé heap

*Notez qu'on peut aussi mapper des fichiers dans l'espace d'adressage de l'app avec une API « **`mmap()`** » / « **`munmap()`** » (pas dans cette version)*



Les données du noyau

Variables globales

Le noyau dispose aussi des variables globales définies dans le code du noyau, allouées lors du chargement du noyau en mémoire.

➡ Elles sont dans le segment kdata, nombre et taille fixe

Variables locales

Chaque thread dispose d'une pile kernel pour la sauvegarde des contextes de fonctions quand il exécute du code kernel (syscall ou ISR)
Ces piles sont créés au moment de la création des threads,
mais il n'y a pas d'allocateur de piles.

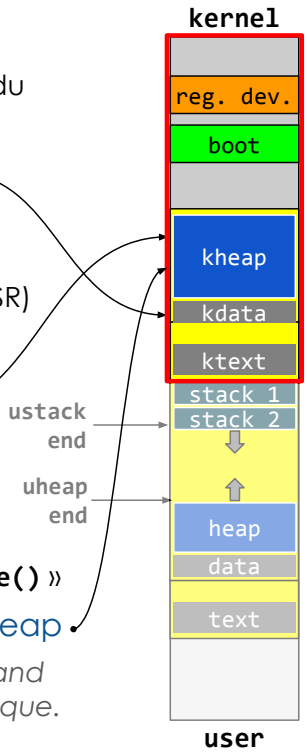
➡ Les piles kernel sont créés dans les objets du noyau

Variables dynamiques

Le noyau gère l'allocation des piles et la position de **`uheap_end`**
il doit aussi allouer dynamiquement des objets pour son compte ou
pour le service des applications, avec une API « **`kmalloc()`** » / « **`kfree()`** »

➡ Il faut un allocateur d'objets dans un segment nommé kheap

Ici, il n'y a qu'un seul espace d'adressage (1 seul processus), mais quand il y en a plusieurs, le kernel gère aussi l'allocation de la mémoire physique.

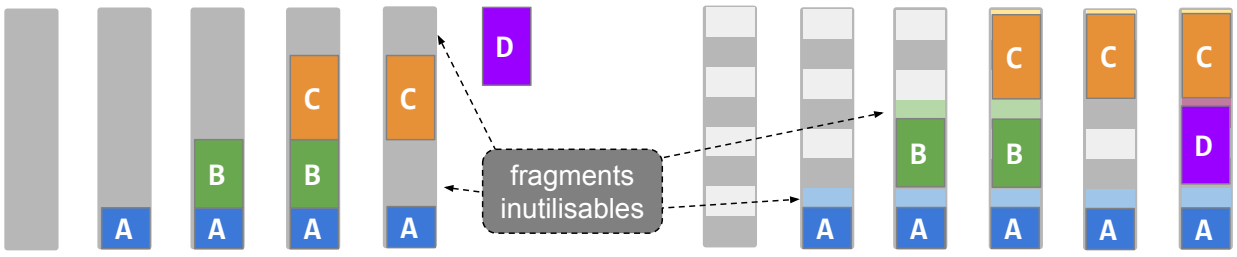


Fragmentation externe et interne

L'allocation dynamique d'espace mémoire doit gérer un problème de perte de place, c'est-à-dire la création involontaire de segments d'adresses inutilisables, c'est un **problème inévitable** de fragmentation d'espace : il en existe deux types

① La zone grise est la zone d'allocation.
On alloue **A**, **B**, **C** puis on libère **B**, et
on veut allouer **D**, mais c'est impossible car les segments libres restants sont trop petits.
Pourtant il reste assez de place libre !

② La zone d'allocation est découpée en cadres.
On alloue **A**, **B**, **C** en les alignant sur les cadres
quand on libère **B**, là on peut allouer **D**, car il reste assez de cadres libres, mais on alloue des multiples de cadres, il y a de la place perdue



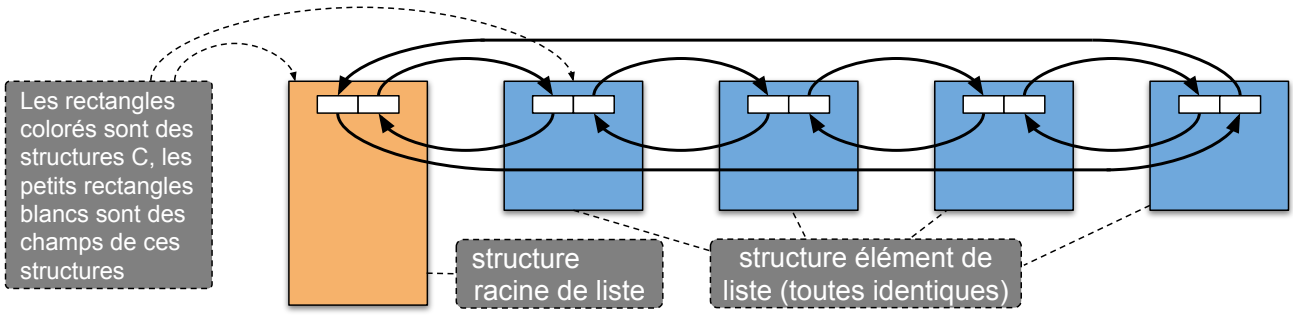
Perte par **fragmentation externe**

Perte par **fragmentation interne**

API de gestion de listes

Gestion de listes chaînées

Le noyau a besoin de gérer des listes chaînées de structures



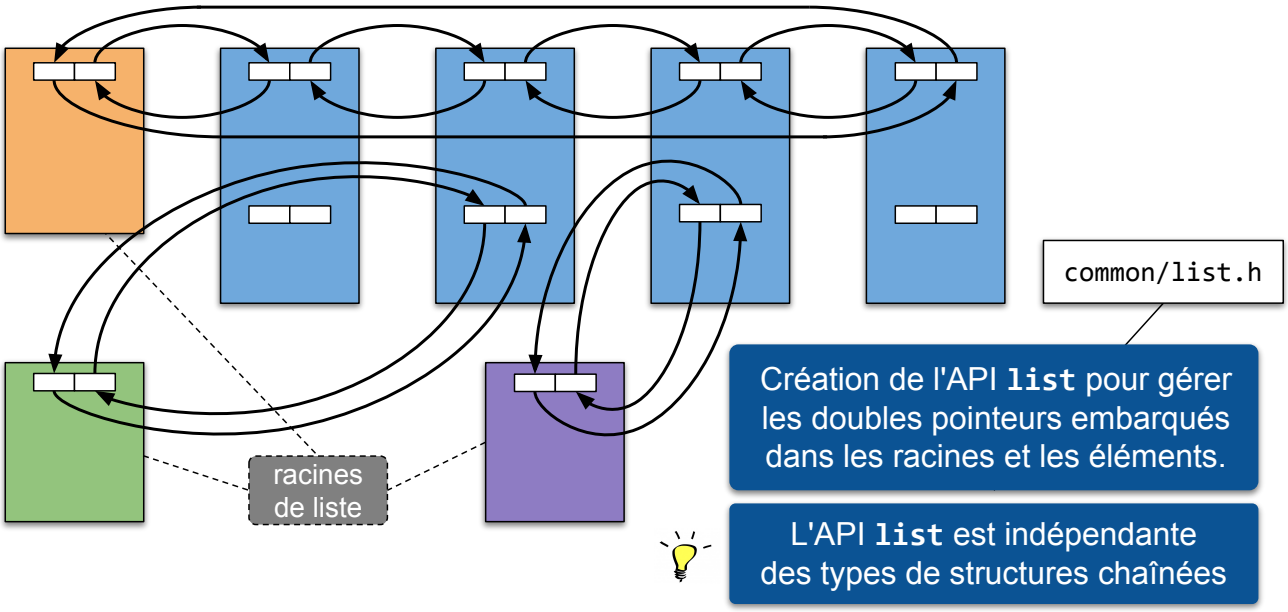
Les besoins sont :

- insérer un élément au début, à la fin ou au milieu
- extraire un élément au début, à la fin ou au milieu
- parcourir tous les éléments
- compter le nombre d'éléments

Le double chaînage permet d'insérer ou d'extraire sans avoir à parcourir la liste

Gestion de listes chaînées

Les éléments de listes doivent pouvoir appartenir à plusieurs listes



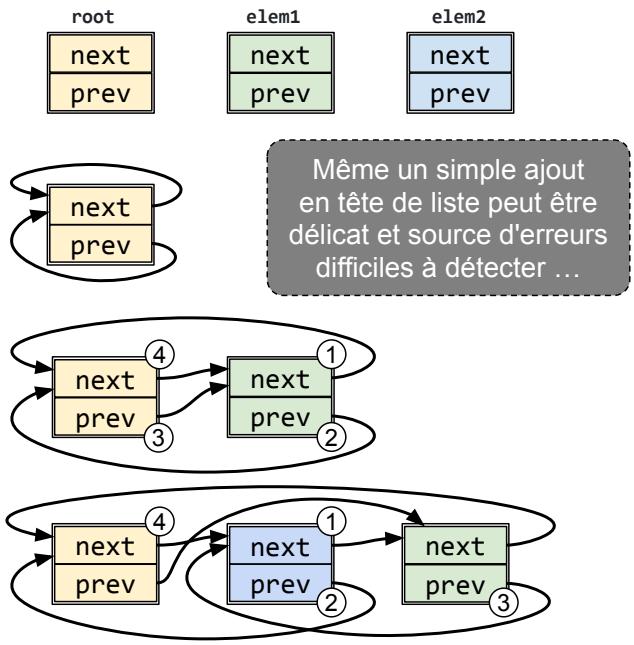
API **list** principe

Exemple de gestion des listes chaînées de structures

```
typedef struct list_s {
    struct list_s * next;
    struct list_s * prev;
} list_t;

void list_init (list_t * root) {
    root->next = root->prev = root;
}

void list_addfirst (list_t * root, list_t * added_item){
 (1) added_item->next = root->next;
 (2) added_item->prev = root;
 (3) root->next->prev = added_item;
 (4) root->next = added_item;
}
```

API exemple

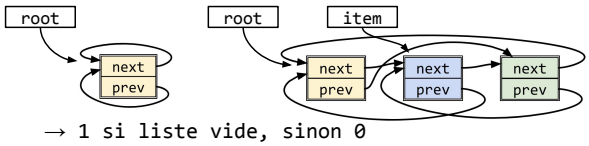
```
list_t root;
list_t elem1, elem2;
bloc d'instructions {
    list_init (&root);
    list_addfirst (&root, &elem1);
    list_addfirst (&root, &elem2);
}
```



API **list** (utilisable par kernel & user)

```c
typedef struct list_s {
    struct list_s * next;
    struct list_s * prev;
} list_t;
```

common/list.h



test

```c
unsigned list_isempty(  list_t * root)                  → 1 si liste vide, sinon 0
unsigned list_isfirst(  list_t * root, list_t * item)   → 1 si item est en tête
unsigned list_islast(   list_t * root, list_t * item)   → 1 si item est en fin
```

consultation

```c
list_t * list_first(    list_t * root)                  → rend un pointeur sur le premier item
list_t * list_last(     list_t * root)                  → rend un pointeur sur le dernier item
list_t * list_next(     list_t * item)                  → rend un pointeur sur l'item suivant
list_t * list_prev(     list_t * item)                  → rend un pointeur sur l'item précédent
list_foreach(           list_t * root, list_t * item)
list_item(              list_t * item, typeof item_s, member)
```

def. expliquées slides suivant

modification

```c
void list_init(         list_t * root)                  → initialise les pointeurs
void list_addfirst(     list_t * root, list_t * added_item) → ajoute un item en tête
void list_addlast(      list_t * root, list_t * added_item) → ajoute un item en fin
list_t * list_unlink(   list_t * item)                  → détache un item
list_t * list_getfirst( list_t * root)                  → détache le premier item d'une liste
void list_replace(      list_t * olditem, list_t * newitem) → change un item par un autre
void list_addsort(      list_t * root, list_t * added_item, int (*cmp)(list_t * A, list_t * B) )
                                                        → ajoute un item dans l'ordre défini par cmp
```

l'ajout au milieu est fait avec **list_addfirst()** avec root = item

miscellaneous

```c
unsigned list_nbobj(    list_t * root)                  → compte le nombre d'item
```


API **list** itérateur

Créer des listes c'est bien, il faut pouvoir les parcourir facilement → **list_foreach()**

Usage

```c
bloc d'instructions {
    instructions-avant
    list_foreach (root, item) {          // root est un pointeur sur une list_t
        instructions du corps de boucle  // item est aussi un pointeur sur une
    }                                    // list_t mais créé pour ce foreach
    instructions-après
}
```

- **root** est un pointeur sur une liste d'éléments, **list_foreach** va parcourir cette liste et affecter à chaque tour de boucle la variable **item** à un nouvel éléments de la liste.
- **item** est une variable locale déclarée implicitement par **list_foreach**

Définition

- **list_foreach (root, item)** n'est pas une fonction c'est un **#define**… qui est remplacé par :

```c
list_t * _NEXT, * item;
for ( item = root->next, _NEXT = root->next; item != root; item = _NEXT, _NEXT = _NEXT->next )
```

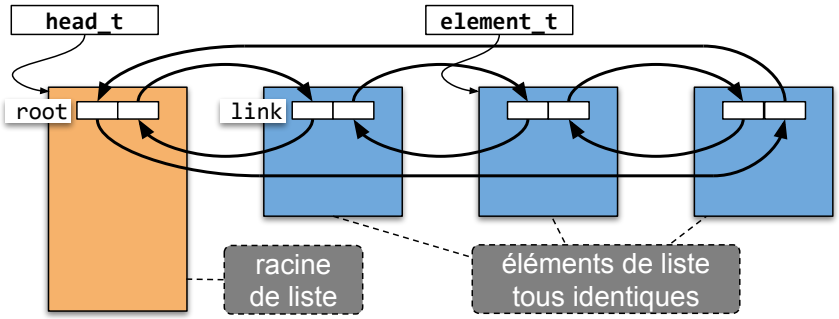
- Si on a besoin de plusieurs **list_foreach** dans un même bloc d'instructions (c'est rare), il faut ajouter des **accolades** à cause de la déclaration des variables locales **_NEXT** et **item**.

```c
{ list_foreach (root, item) { instructions du corps de boucle } }
```

API **`list`** structure porteuse

Il faut pouvoir retrouver le pointeur sur la structure porteuse → **`list_item()`**

- La structure **`list_t`** est faite pour être "**embarquée**" dans une structure porteuse,
- **Ce sont les structures `list_t` qui sont chaînées entre elles**, mais on doit pouvoir calculer l'adresse des structures qui les embarque
- `rout_t * racine = ...`
  `list_foreach ( &(racine->root), item)`




```c
typedef struct head_s {
    [...]
    type_Z * field_Z;
    list_t root;
    type_T * field_T;
    [...]
} head_t;
```

```c
typedef struct element_s {
    [...]
    type_X * field_X;
    list_t link;
    type_Y * field_Y;
    [...]
} element_t;
```

`list_item ( item, element_t, link )` →
- rend le pointeur sur la structure **`element_t`**
- le pointeur sur la structure : **`list_t`**
- le type C de l'élément porteur : **`element_t`**
- et le nom du champ **`list_t`** dans **`element_t`**

Exemple complet d'usage de l'API list

```
$ ./list
Entrez votre nom : Franck
family Archi2 : Franck, Monique, Jean-Claude
$
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <list.h>

typedef struct family_s {
    char *lastname;
    list_t root;
} family_t;

typedef struct person_s {
    char *firstname;
    list_t item;
} person_t;

char * input (char * mess) {
    char buf[256];
    fprintf (stdout, mess);
    fgets(buf, sizeof(buf), stdin);
    buf[strlen(buf)-1] = 0;// del \n
    return strdup (buf);
}
```

includes Linux, mais pour kO6, il faut juste **`libc.h`** ou **`klibc.h`**

les deux structures seront chaînées : `family_t` : racine `person_t` : élément

~ input de Phyton :-)

```c
int main () {
    family_t archi2;
    person_t jean_claude;
    person_t monique;

    archi2.lastname = "Archi2";
    jean_claude.firstname = "Jean-Claude";
    monique.firstname = "Monique";

    person_t * moi = malloc (sizeof(person_t));
    moi->firstname = input ("Entrez votre nom : ");

    list_init (&archi2.root);
    list_addlast (&archi2.root, &monique.item);
    list_addlast (&archi2.root, &jean_claude.item);
    list_addfirst (&archi2.root, &moi->item);

    printf ("family %s : ", archi2.lastname);
    list_foreach (&archi2.root, i) {
        person_t *ptr = list_item (i, person_t, item);
        printf ("%s, ", ptr->firstname);
    }
    printf ("\b\b \n");
    return 0;
}
```

déclaration et initialisation

création de la liste

parcours de la liste

Allocateur de piles user

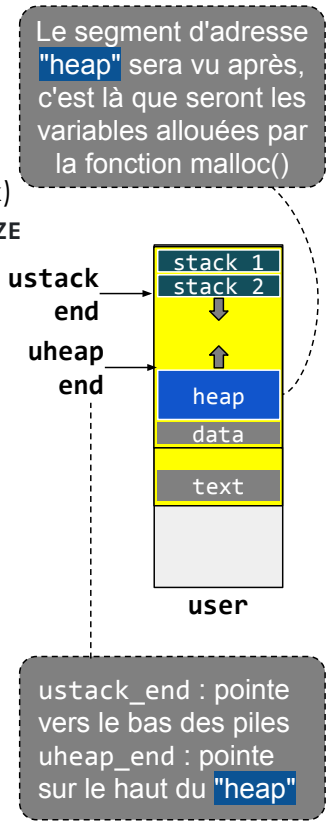
Besoins et Principe

Besoins

- Chaque thread a besoin d'une pile dans l'espace user (nommée `stack`)
  Ici, toutes les piles auront la même taille décidée à l'avance `USTACK_SIZE`
- L'allocation est faite par le noyau lors de la création d'un thread
- Les piles sont placées en haut du segment `.data`
- Il faut un moyen de vérifier que les piles n'ont pas débordé

Principe

- Le noyau gère une **liste de piles vides**, au départ cette liste est vide
- L'allocation d'une pile consiste à prendre la première pile vide
  → S'il n'y en a pas, le noyau descend le pointeur `ustack_end` à la condition qu'il n'entre pas en collision avec le `heap` (`>uheap_end`)
- La libération d'une pile consiste à la remettre la liste des piles vides
  → La liste des piles est triée par adresse croissante
  → Si on libère la pile placée juste au dessus de `ustack_end`, alors on peut remonter `ustack_end` et on parcourt les piles libres (elles sont triées) pour libérer celles qui sont les plus basses.



Allocation (toutes les piles ont la même taille)

```
int * malloc_ustack (void)
{
    int * top;
1   int * end = (int *)list_getlast (&FreeUserStack);
    if (end == NULL) {
2       top = _user_mem.ustack_end;
        end = top - USTACK_SIZE/sizeof(int);
        PANIC_IF (end < _user_mem.uheap_end,
            "no more space for user stack!\n");
        _user_mem.ustack_end = end;
3   } else {
        top = end + USTACK_SIZE/sizeof(int);
    }
4   top--;
    *top = *end = MAGIC_STACK;
    return top;
}
```

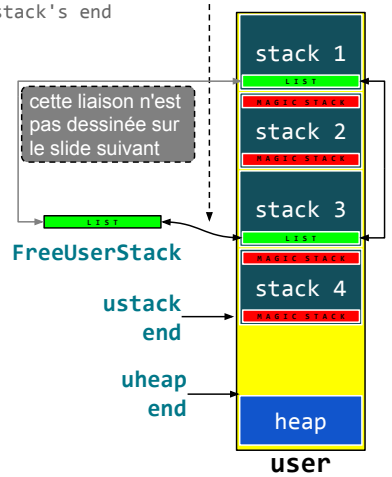
```
// top will be the new stack pointer
// get last free stack (biggest addr)
// if there is no more free stack
// try to get one
// and compute the end of the stack
// if the stack end is in the heap
// it is impossible to solve that
// expand the stacks' region

// compute stack's top from stack's end

// get a word to put MAGIC
// to be able to check free
// finally return the top
```

PANIC_IF(cond,str,...) vue à la fin de ce cours

Commentaires du code il y en a beaucoup !

la liste **FreeUserStack** est une liste doublement chainée circulaire mais on représente juste avec ←→ pour la lisibilité du schéma

cette liaison n'est pas dessinée sur le slide suivant

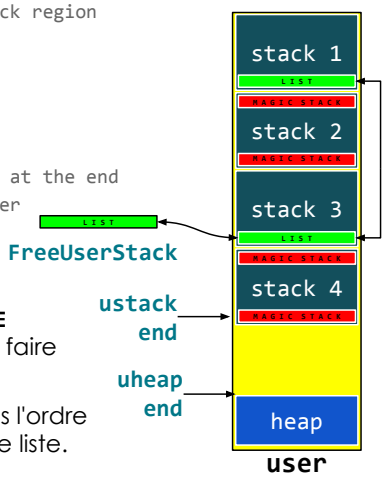


1. **list_t FreeUserStack** est la racine des piles libres, elles sont triées par adresses croissantes, on prend la dernière (ici **stack1**)
   Le but est d'utiliser les piles hautes pour espérer libérer les piles basses.
2. S'il n'y a pas de pile libre, on descend **ustack_end** de **USTACK_SIZE** (taille de pile) et si on arrive en dessous de **uheap_end** c'est qu'il n'y a plus de place → **panic**
3. Calculer le haut de pile
4. Ecrire le nombre **MAGIC_STACK** en haut et en bas et rendre le haut de pile

Libération

```
static int cmp_addr (list_t * curr, list_t * new) {return (int)(curr - new);}

void free_ustack (int * top)
{
1   int * end = 1 + top - USTACK_SIZE/sizeof(int);          // last int of ustack (+1 because of MAGIC)
    PANIC_IF (*top != MAGIC_STACK, "Corrupted top Stack");  // if no magic number then panic
    PANIC_IF (*end != MAGIC_STACK, "Corrupted end Stack");  // if no magic number then panic
    if (end ==_user_mem.ustack_end) {                       // if it is the lowest stack
        _user_mem.ustack_end += USTACK_SIZE/sizeof(int);    // shrink the stacks' region
        list_foreach (&FreeUserStack, stack) {              // foreach free stack
            if ((int *)stack != _user_mem.ustack_end)       // if it isn't the end of stack region
2               break;                                      // then stop trying to shrink
            end = (int *)list_getfirst (&FreeUserStack);    // extract the stack
            end += USTACK_SIZE/sizeof(int);                 // new end of stacks'region
            _user_mem.ustack_end = end;                     // save this new end
        }
3   } else                                                  // else the freed stack isn't at the end
        list_addsort (&FreeUserStack,(list_t*)end,cmp_addr);// add it in free list in order
}
```

cmp_addr() est utilisée par **list_addsort()** pour comparer 2 items de liste, **curr** pointera successivement sur tous les items de la liste et **new** pointe sur l'item à ajouter, **list_addsort()** ajoute **new** ajoute avant **curr** dans la liste si **cmp_addr(curr,new) ≥ 0**
Attention : la liste doit déjà être triée



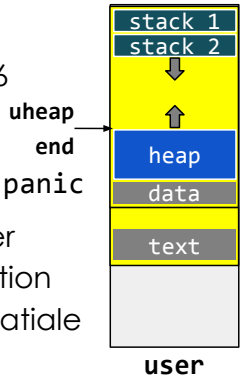
1. calcul de **end** : la fin de la pile et vérification d'absence de corruptions (**top** / **end**)
2. Si on libère la dernière pile (ici **stack 4**) alors remonter **ustack_end** de **USTACK_SIZE** puis parcours de la liste triée des piles libres et si la première est tout en bas, il faut faire remonter **ustack_end** (ici **stack 3** mais pas **stack 1** parce qu'elle n'est pas en bas)
3. Si la pile libérée n'est pas en bas on la remet dans la liste des piles libres, mais dans l'ordre des adresses croissantes grâce **cmp_addr** qui définit la relation d'ordre des items de liste.

Allocateur first fit

Allocation d'objets pour l'application

- L'application a besoin d'allouer de la mémoire de taille quelconque pour ses variables dynamiques de quelques octets à plusieurs dizaines de méga-octets.
- Pour ça, la libc (utilisateur) gère un segment d'adresse nommé tas (heap en anglais) dans lequel, elle alloue et libère ces variables à la demande avec les fonctions
  - `void * malloc (size_t size)` → rend un pointeur sur l'objet ou NULL
  - `void free (void * ptr)` → prend un pointeur alloué par `malloc()`
- La libc peut demander au noyau de changer la taille du tas avec les appels systèmes `brk()` et `sbrk()` qui déplacent le pointeur **`uheap_end`** de fin de tas (https://man.developpez.com/man2/brk/), mais pour kO6
  - `void * sbrk (int incr)` → incr peut être négatif

  Si c'est impossible, collision avec les piles ou la section data → kernel `panic`



- Les objets alloués sont alignés sur des lignes de cache pour : (1) éviter les faux partages (pb cohérence de cache), (2) limiter la fragmentation externe et (3) améliorer l'efficacité des caches - de par la localité spatiale

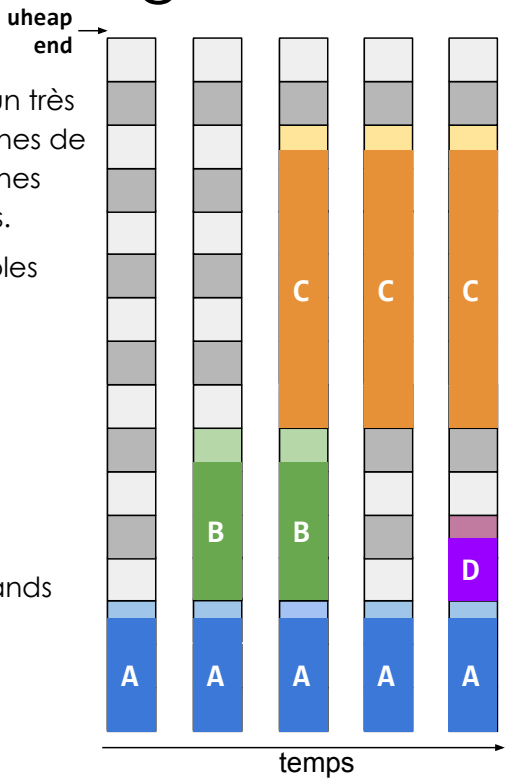
Politiques de remplissage

- A gauche sont représentés des états successifs du tas.
- Chaque case représente une ligne de cache (ici, c'est un très petit tas) puisque les objets alloués sont alignés sur les lignes de cache pour éviter qu'une ligne soit présente dans 2 caches alors que les données s'y trouvant ne sont pas partagées.
- Lors de la recherche de place, plusieurs politiques possibles

  First Fit choix de la première zone libre assez grande pour répondre à la demande.

  Next Fit politique First fit mais en commençant là où on a fait la dernière allocation.

  Best Fit Recherche de la zone dont la taille est la plus proche de la taille demandée
- First Fit / Next Fit sont plus rapides mais font perdre les grands segments, Next Fit permet d'étaler les objets dans le tas.
- Best Fit est plus lent, mais il préserve les grands segments, toutefois il engendre plus de petits segments.



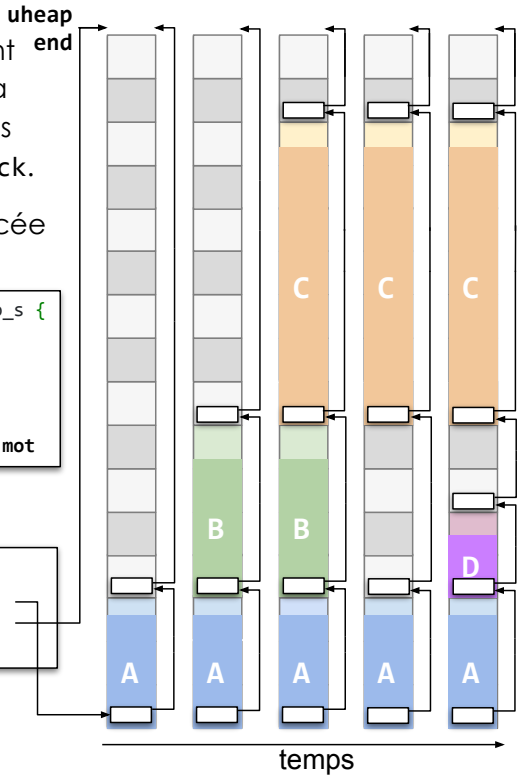
Architecture interne du tas

- Pour décrire l'algorithme, on nomme `"block"` un segment d'adresse alloué ou vide. Dans le schéma de droite, il y a 6 états successifs du tas : d'abord 2 `blocks`, puis 3 et dans le dernier état, 5 `blocks`. Au tout début, il y a un seul `block`.
- une structure d'information nommée `block_info` est placée au début de chaque `block`, elle contient :
  - L'état du `block` (full / free)
  - La taille du `block` en nombre de lignes de cache
  - Un nombre `MAGIC` pour tester l'absence de corruption

```
typedef struct block_info_s {
    unsigned full:1;
    unsigned magic:7;
    unsigned size:24;
} block_info_t;

Le C compacte ça dans un mot
```

- La structure racine minimale
  - Pointeurs sur les `block_info` le plus bas et le plus haut
  - Pour l'algorithme Next Fit, il faudrait un autre pointeur pointant le dernier `block` alloué.

```
static struct heap_s {
    block_info_t *beg;
    block_info_t *end;
} Heap;
```



Principe de l'algorithme First Fit

```
void * malloc (size_t size);
```

1. Parcours du tas depuis le premier `block` jusqu'à trouver un `block` libre assez grand.
2. Si la fin du tas est atteinte sans trouver de place alors **`merger`** les `blocks` libres et retenter 1 fois.
3. Si un `block` libre est trouvé, mais trop grand alors le scinder en deux : le 1er `full`, le 2nd `free`
4. Mettre à jour les `block_info` (état, taille, nombre `MAGIC`)
5. Rendre l'adresse qui suit juste `block_info` (adresse de `block_info + sizeof(block_info_t)`)

```
void free (void * ptr);
```

1. Trouver le `block_info` qui doit se trouver à l'adresse (`ptr - sizeof(block_info_t)`)
2. Vérifier la présence du nombre `MAGIC`, sinon c'est que le tas a été corrompu → `exit(error)`
3. Changer l'état du block à `free` (on ne fait pas d'union avec les `blocks` libres voisins).

```
static void merge (block_info_t * ptr);
```

1. `merge()` est utilisée uniquement par `malloc()` lorsqu'elle est n'a pas trouvé de place
2. Parcours de tout le tas depuis le tout premier `block` (reçu en argument)
3. Pour chaque `block free`, parcours de tous les suivants jusqu'à trouver un `block full`
   faire la somme des tailles des `blocks free` rencontrés et mettre à jour le premier `block_info`

Conclusion et amélioration

- Ce qu'il faut comprendre, c'est que la gestion du tas de l'application est la responsabilité de la libc, le noyau ne fait que donner de la place avec l'appel système `sbrk(incrément)` (POSIX définit aussi `brk(address)`)
- L'utilisateur peut implémenter les algorithmes qu'il veut en fonction des profils d'usage de son application pour réduire la latence et la fragmentation.
- Les algorithmes First/Next/Best Fit ne sont pas les plus efficaces mais ils n'imposent pas ou peu de contraintes d'usage et ils sont simples.
- Si un programme gère des millions d'objets de même taille et qu'il fait beaucoup d'allocations et de libération de ces objets, il peut créer des listes d'objets libres dans lesquels il peut piocher (allouer) et remettre (libérer) facilement.
- D'autres part, les objets alloués par malloc sont alignés sur des lignes de caches pour éviter les faux partages. Si 2 variables totalement indépendantes sont dans la même ligne mais qu'elles sont utilisées par des processeurs différents, alors la même ligne est dans les 2 caches et il y a des fausses demandes de cohérence.

Allocateur slab pour les objets du noyau

Remarques préliminaires

ICI, le noyau de kO6 gère une seule application et donc un seul espace d'adressage.

> Le noyau n'a pas un espace d'adressage physique à partager entre plusieurs applications. Pour avoir plusieurs applications simultanément en mémoire, il faudrait que le SoC dispose d'une unité de gestion de la mémoire (nommé MMU pour Memory Management Unit) pour offrir à chaque application, un espace d'adressage virtuel mappé dans l'espace d'adressage physique par le noyau.

Ici, le noyau doit juste allouer 2 types d'objets, pour lui et pour l'unique application

1. Des petits objets (≤ `4kio`) pour lui :
   → struct : threads, devices, inodes, mutex, etc.
2. Des gros objets (> `4kio`) pour lui ou pour l'application :
   → piles user, caches de fichiers, buffers réseau, buffers partagés, mapping de fichiers, etc.

Nous allons limiter les gros objets aux piles users (avec l'allocateur déjà présenté), pas de mapping de fichiers (dû à l'absence de MMU), pas de buffers partagés (car une seule application), les autres objets seront limités à `4kio` pour le noyau.

Allocateur d'objets

Le noyau a juste besoin d'un allocateur d'objets de taille ≤ `4kio`

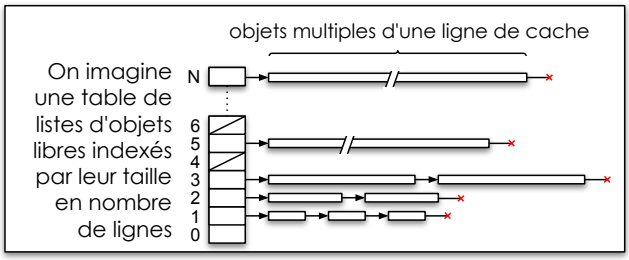
- Il propose une API « **`ptr = kmalloc(size)`** » / « **`kfree(ptr)`** » semblable à celle de la libc
- Cet allocateur vise à **réduire au maximum la fragmentation externe** (la pire des deux) et **il doit être très rapide**, ainsi il n'est pas raisonnable de rechercher dans des listes

L'idée, c'est que le noyau alloue et libère toujours des objets de taille fixe, alors il faut :

➔ Créer des listes d'objets libres de taille fixe et multiples d'une ligne de cache pour éviter les faux partages

➔ Pour allouer, if suffit d'extraire le 1er élément de la liste d'objets de la bonne taille

➔ Pour libérer, il suffit de replacer l'objet en tête de la liste d'objet de cette taille



on peut faire différemment, cette solution a le mérite d'être compacte et rapide

Les problèmes à résoudre

- Quand et comment créer les listes d'objets libres, avec peu de fragmentation externe ?
- Comment adapter le nombre d'objets libres au besoin du noyau ?

Concept de **slab** / dalle

La taille des objets est un multiple d'une ligne de cache et au plus 1 page de `4kio`

- Si la ligne de cache fait 16 octets (4 mots), alors l'objet le plus petit fera 16 octet.

Pour limiter la fragmentation, tous les objets de même taille sont alloués dans des grands segments de taille fixe que l'on nomme **slab** (dalle en français)

- Les dalles (slab) ont une taille de 2^n pages de `4kio` (1, 2, 4, etc. pages)

Pour 1 ligne de cache de 16 octets (2^4) dans un **slab** d'1 page (`4kio` = 2^{12} octets),

- Il y a 256 (2^8) objets d'une ligne (ou moins d'une ligne)
- Il y a 128 (2^7) objets de 2 lignes (ou entre 1 et 2 lignes)
- Il y a 85 objets de 3 lignes (`4096/3*16=85` mais `85*16*3=4080` donc 16 octets perdus…)
- etc.

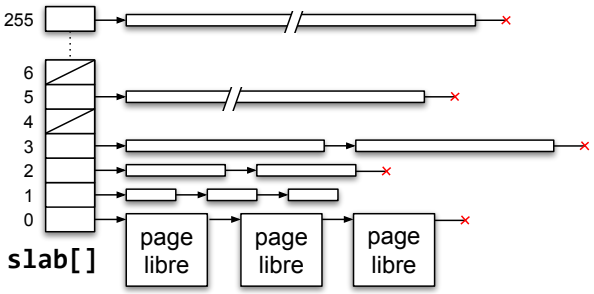
On définit le numéro d'un slab comme le nombre de lignes de ses objets (de 1 à 256)

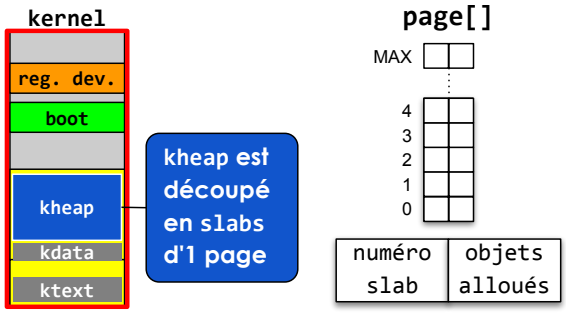
Pour les objets inférieurs à ⅛ pages (512 octets), les **slab**s font 1 seule page, mais au delà, les **slab**s sont en principe plus grands pour réduire la fragmentation externe des **slab**s, MAIS, dans l'état actuel de kO6, les **slab**s font toujours une seule page !

Pour information, les **slab**s sont normalement alloués par un autre allocateur nommé "**buddy allocator**" qui alloue de **2^n** pages, mais nous le verrons pas ici…

Principe de l'algorithme implémenté

- Les **slab** font toujours 1 et 1 seule page de **4kio**
- Les objets sont tous des multiples d'1 ligne de cache **mais au minimum 16 octets** (4 mots)
- Il y a donc au plus 256 objets d'1 ligne (16 o) / **slab**
- On définit la variable globale **slab[]**, une table qui contient les racines des listes d'objets libres indexés par leur taille (en nb de lignes)
  - **slab[i]** chaîne les objets de **i** lignes
  - **slab[0]** chaîne les pages libres
- L'objet le plus grand (**4kio**) fait au plus 256 lignes la table **slab[]** a donc au plus 256 cases
- A l'initialisation, **kheap** est découpé en **slab** d'1 page chaînées entres elles et attachées à **slab[0]**





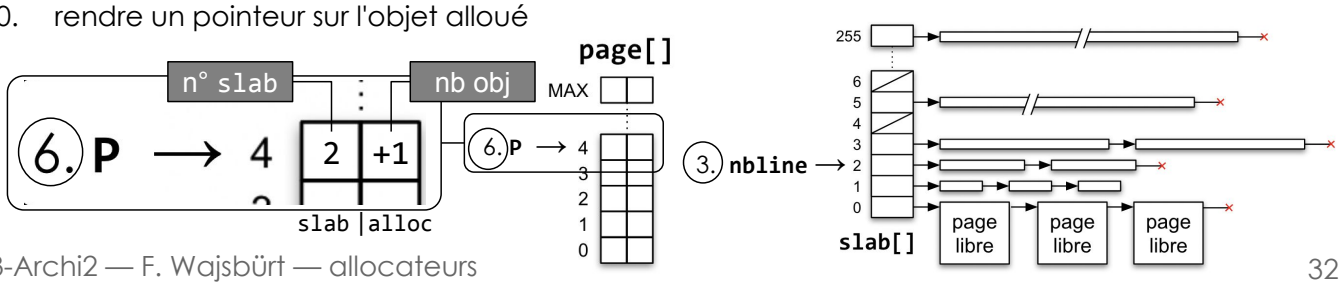
- On définit la table **page[]** contenant les descripteurs de toutes les pages de **kheap**. Pour chaque page, on enregistre à quelle liste de **slab** elle appartient (de 0 à 255, 0 pour les **slabs** de 1 page) et combien d'objets y sont alloués (de 0 à 255, 0 == 256 sauf pour le **slab[0]**)

Allocation

```
void * kmalloc (size_t size);
```

1. Si `size` est supérieur à 1 page (> 4kio) → `panic "trop grand"`
2. Si `size` est égale à 1 page (== 4kio) mais que **slab[0]** est vide → `panic "plus de place"`
3. `nbline` ← le nombre de lignes de `size` (arrondi par excès) pour trouver la liste où rechercher
4. Si `size` est inférieure à 1 page (< 4kio) et que `slab[nbline]` est vide, alors
   - Allouer une page vide `P` par un appel récursif (`kmalloc(4kio)`)
   - Découper cette page `P` en objets de taille `nbline` et les chaîner dans `slab[nbline]`
   - Mettre à zéro le nombre d'objets alloués pour `P` dans la table `page` : `page[P].alloc = 0`
5. Extraire le premier objet de `slab[nbline]` (il y a forcément un objet libre)
6. Déterminer le numéro de page `P` où se trouve l'objet
7. mettre à jour la case `page[P].alloc` avec le nombre d'objets alloués : `page[P].alloc++`
8. Ecrire le numéro de `slab` utilisé pour cette page P : `page[P].slab = nbline`
9. Initialiser l'objet à `0` (*c'est une sécurité, on le fait pour la libc avec la fonction* `calloc()`)
10. rendre un pointeur sur l'objet alloué

sera mis à 1 à l'étape 7



Libération

Pour libérer un objet, l'idée c'est simplement de le remettre en tête de la liste correspondant à sa taille dans la table `slab`[], mais si tous les objets d'un même slab sont libres alors, le `slab` complet doit être libéré et rendu à la liste des pages libres, c'est la table `page`[] qui dit si le `slab` est vide.

Un objet alloué est identifié par le pointeur sur le premier octet, ici nous avons besoin de la taille de l'objet pour connaître son `slab`, nous pourrions l'enregistrer quelque part, mais nous allons plus simplement l'indiquer lors de l'appel de `kfree()`, c'est possible car que le noyau sait ce qu'il fait.

```
void kfree (void * obj, size_t size)
```

1. `nbline` ← le nombre de lignes de `size` (arrondi par excès) pour trouver la liste où rechercher
2. Déterminer le numéro de page `P` où se trouve l'objet
3. Si `nbline` > 255 ou si `P` n'est pas dans `kheap` alors l'objet n'est pas correct → `panic "bad arg"`
4. Ajouter l'objet obj dans `slab[nbline]`
5. Si `size` est égale à `1 page` alors c'est fini, sortir de `kfree()`
6. Décrémenter le nombre d'objets alloués dans le `slab : page[P].alloc--`
7. Si le nombre d'objets alloués est tombé à 0 alors
   - Parcourir tous les objets `obj` de `slab[nbline]` et retirer de la liste ceux appartenant à `P`
   - Marquer le fait que la page P n'est plus utillisée `page[P]=0`
   - Ajouter la page P dans `slab[0]`

à faire tout de suite parce que 0 == 256

Un peu de code

- Nous n'allons pas lire tout le code ici ! Mais vous pouvez le lire dans `kernel/kmemory.c` 😉
- Toutefois, nous allons regarder le code de `memory_init()`
- Pour chainer les objets, on utilise évidemment l'API list vue au début du cours

kmb : kernel memory begin
kme : kernel memory end

parce qu'on limite la table page[] à une seule page

```
void memory_init (void)
{
    CacheLineSize = CEIL(cachelinesize(),16);
    NbPages = (kme-kmb)/PAGE_SIZE;
    MaxLinePage = PAGE_SIZE / CacheLineSize;

    PANIC_IF (NbPages*sizeof(page_t) > PAGE_SIZE,
        "kmalloc can't handled %d pages", NbPages);

    list_init (&FreeUserStack);

    for (int i = 0 ; i < MaxLinePage ; i++)
        list_init (&Slab[i]);
    for (char *p = kmb; p != kme; p += PAGE_SIZE)
        list_addlast (&Slab[0], (list_t *)p);
    Page = kmalloc (PAGE_SIZE);
    kmalloc_test(1000,PAGE_SIZE/8);
}
```

```
    // true line size, but expand to 16 min
    // maximum number of pages
    // 256 when line is 16, 128 for 32, etc.

    // Not more than 2048 thus 8MB
    // write a message then panic

    // initialize the free user stack list

    // initialize each list is Slab table
    // Slab[i] -> i*cachelinesize objects
    // initialize the page (slab) list
    // pointed by Slab[0]
    // allocate the Page descriptor table
```

fonction d'autotest optionnelle

initialisation des liste de stack users et de la liste des page libres dans `slab[0]`

Conséquence sur les threads

Besoins

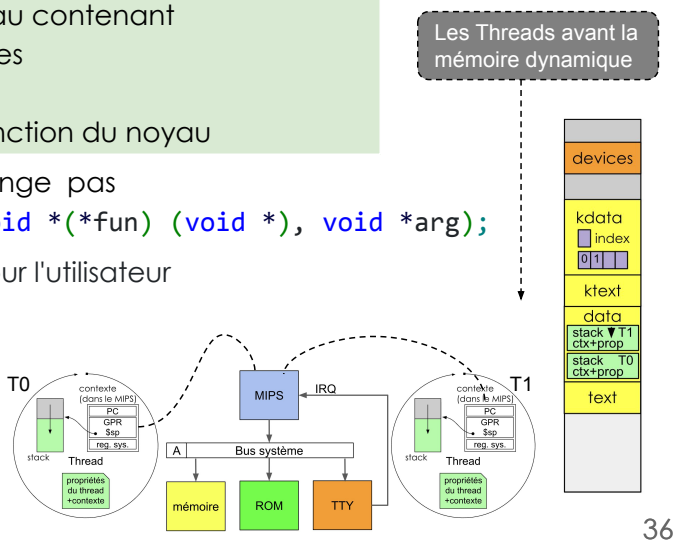
- Pour gérer chaque thread, nous **avions** créé une variable globale dans l'espace d'adressage utilisateur pour la pile, la table de sauvegarde des registres et les propriétés. Mais, c'était avant l'allocation de mémoire.
- Désormais pour chaque thread, nous allons créé
  - une pile dans l'espace utilisateur
  - une structure thread dans l'espace noyau contenant
    - la table de sauvegarde des registres
    - les propriétes
    - et une pile pour l'exécution des fonction du noyau
- L'API utilisateur de création de thread ne change pas

```
int thread_create (thread_t * thread, void *(*fun) (void *), void *arg);
```

- Ce qui change c'est que le type `thread_t` pour l'utilisateur

```
typedef unsigned thread_t;
```

ce type devient un type « *opaque* »
Le noyau va créer les structures et initialiser `thread` avec un identifiant

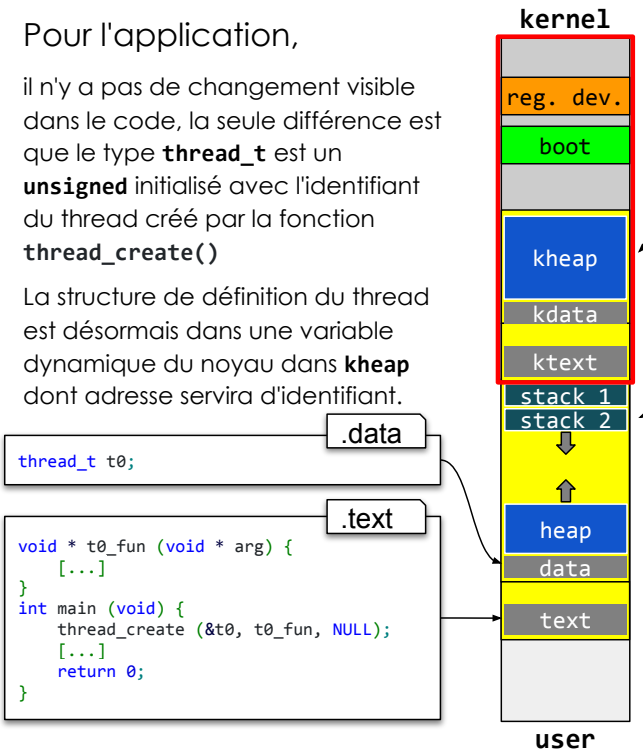


Structure kthread_t et les piles de thread

Pour l'application,

il n'y a pas de changement visible dans le code, la seule différence est que le type **thread_t** est un **unsigned** initialisé avec l'identifiant du thread créé par la fonction **thread_create()**

La structure de définition du thread est désormais dans une variable dynamique du noyau dans **kheap** dont adresse servira d'identifiant.



kheap

```
typedef struct kthread_s {
    int state;                      // thread state (RUNNING, etc.)
    int start;                      // ptr to fct which calls fun(arg)
    int fun;                        // ptr to the thread function
    int arg;                        // thread argument
    int tid;                        // thread identifier
    int context[TH_CONTEXT_SIZE]; // table to store CPU registers
    int * ustack_b;                 // user stack beginning
    int * ustack_e;                 // user stack end
    int * kstack_b;                 // kernel stack beginning
    int kstack[1];                  // kernel stack end
} kthread_t;
```

Pour le noyau,

une variable de type **kthread_t** est allouée pour chaque thread dans une page de 4kio de **kheap**

La structure **kthread_t** a les mêmes informations que l'ancienne structure **thread_t**, sauf qu'il y a **deux** piles :

- Une pile **user** allouée dans la zone des stacks user, avec **ustack_b** et **ustack_e** pointant resp. sur le début et sur la fin de la pile (le haut et le bas)
- Un pile kernel dans la structure **kthread_t** avec **kstack_b** et **kstack** resp. le début et la fin, le début étant la fin de la page utilisé par la struct **kthread_t**

Changement de pile : User → Kernel

- Dédoublement du fichier thread.h (user & kernel)
  - l'application utilisateur n'a plus accès à la structure qui définit le thread
  - la structure kthread_t contient tout dont les pointeurs sur la pile user
- changement de pile
  - le code noyau utilise la pile kernel du thread qui fait un appel système ou qui traite une ISR, il faut faire la commutation de pile à l'entrée de kentry

pas de changement kentry analyse de la cause et saut

```
kentry:
    mfc0    $26,    $13
    andi    $26,    $26,    0x3C
    li      $27,    0x20
    beq     $26,    $27,  syscall_handler
    beq     $26,    $0,   irq_handler
    j       kpanic
```

changement de $29 U→K

syscall n'est utilisé que par l'application, il faut changer de pile, ThreadCurrent pointe sur le thread courant, on en déduit le haut de la pile

```
syscall_handler:
    la      $26,    ThreadCurrent
    lw      $26,    ($26)
    sw      $29,    TH_SIZE-8($26)
    addiu   $29,    $26,    TH_SIZE-10*4
    [...]
    lw      $29,    8*4($29)
    eret
```

Les IRQ peuvent survenir alors le MIPS exécute du code user ou du code kernel, il faut tester le mode d'exécution pour savoir si on doit changer de pile, si oui on exécute le code irq_user

```
irq_handler:
    mfc0    $26,    $12
    ext     $26,    $26,    4,      1
    move    $27,    $29
    beq     $26,    $0,     irq_kernel
irq_user:
    la      $26,    ThreadCurrent
    lw      $26,    ($26)
    addiu   $29,    $26,    TH_SIZE-4
irq_kernel:
    [...]
    lw      $29,    20*4($29)
    eret
```

Les nouveaux fichiers de kO6

```
common
    debug_off.h
    debug_on.h
    list.h
    syscalls.h
    usermem.h
kernel
    harch.c
    harch.h
    hcpua.S
    hcpuc.c
    hcpu.h
    kernel.ld
    kinit.c
    klibc.c
    klibc.h
    kmemory.c
    kmemory.h
    ksyscalls.c
    kthread.c
    kthread.h
    Makefile
Makefile
uapp
    main.c
    Makefile
ulib
    crt0.c
    libc.c
    libc.h
    Makefile
    memory.c
    memory.h
    thread.c
    thread.h
    user.ld
```

Ce que nous avons vu

Nous avons parlé de la nécessité de faire de l'allocation dynamique et des pertes en mémoire à cause du problème de fragmentation interne et externe.

Nous avons vu 3 allocateurs de mémoire dynamiques avec 3 objectifs distincts

1. L'allocateur de pile utilisateur pour les threads. Ici, que des piles de taille identique.
2. L'allocateur de mémoire dynamique pour l'application utilisateur. Cet allocateur utilise une politique first-fit, c'est-à-dire que l'objet est alloué dans le premier segment trouvé avec une taille suffisante. La taille des objets alloué va de 1 ligne de cache à tout l'espace disponible
3. L'allocateur d'objets pour le noyau. Les objets de même taille sont alloués dans des slabs. L'allocateur gère des listes d'objets libres dans lesquelles il est rapide d'allouer et de rendre. La complexité est lorsqu'il faut créer de nouveaux slab ou qu'il faut les libérer.

Nous avons vu un mécanisme de gestion de listes doublement chaînées permettant de créer facilement des listes d'objets, de les parcourir et de les modifier sans manipuler les pointeur de chaînage. Ce mécanisme est utilisé par l'allocateur de pile et l'allocateur slab.

Enfin, nous avons vu ce que permet l'allocation dynamique de mémoire pour la gestion des threads, à savoir la possibilité pour le noyau d'allouer la structure thread et de gérer deux piles par thread, une pour l'application, et une pour l'exécution de son propre code.

En TME

- Comme d'habitude, quelques questions sur le cours
- Programmation / modification des allocateurs user et kernel