

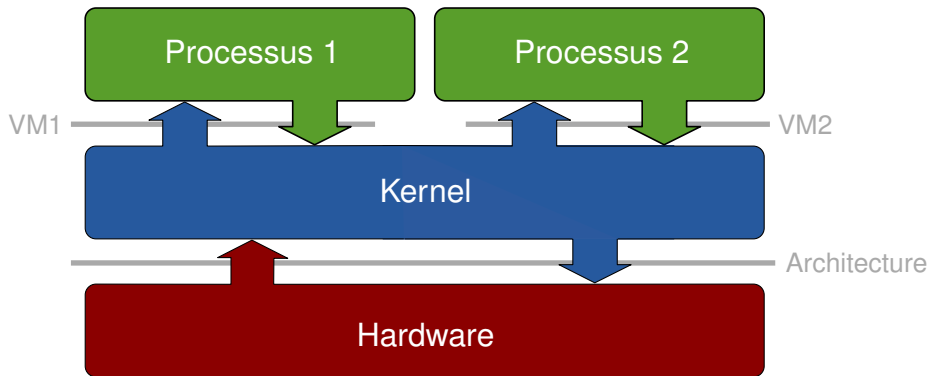
Gestion mémoire 2: allocation dynamique

Guillaume Salagnac

Insa de Lyon – Informatique

2020–2021

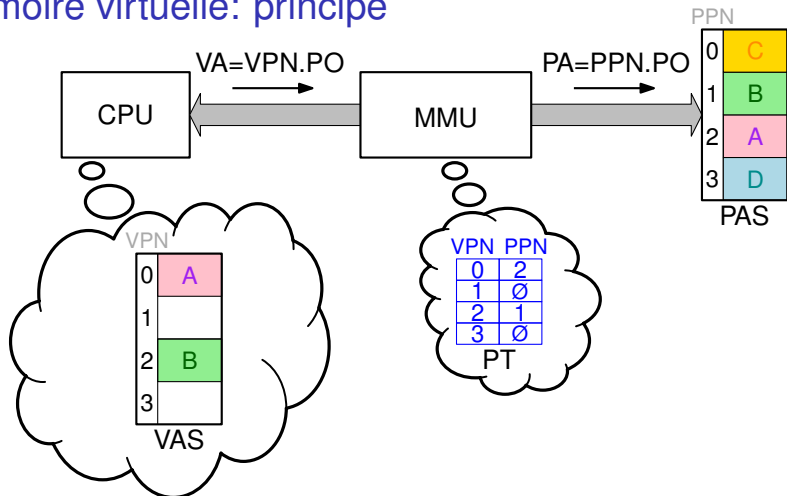
Résumé des épisodes précédents



Le processus vu comme une «machine virtuelle»

- un processeur pour moi tout seul: «CPU virtuel»
- une mémoire pour moi tout seul: «mémoire virtuelle»

Mémoire virtuelle: principe



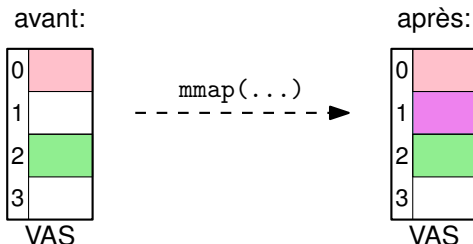
Virtualisation de l'espace d'adressage:

- CPU voit seulement des **adresses virtuelles** VA
- MMU/TLB traduit à la volée les VA en **adresses physiques** PA
- correspondance $VA \rightarrow PA$ indiquée par la **Table de Pages** PT

Mémoire virtuelle: services rendus par le noyau

Quelques-unes des fonctions du noyau:

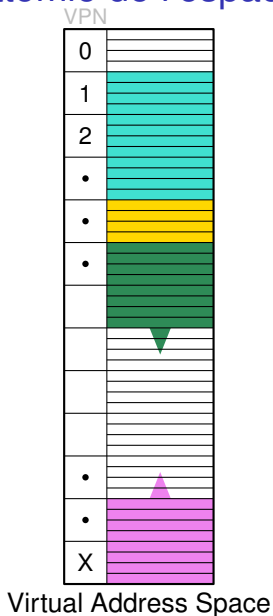
- reconfigurer la MMU à chaque **changement de contexte**
 - un processus = une table de pages
- réagir aux **fautes de page**
 - assurer le va-et-vient (**swap**) des pages entre DRAM et disque
- **allouer** de nouvelles pages à un processus
 - par ex: appel système `mmap()`



Plan

1. Introduction: rappels sur la mémoire virtuelle
2. Allocation dynamique: définition du problème
3. Fragmentation du tas et stratégies d'allocation
4. Techniques d'implémentation et interface utilisateur

Anatomie de l'espace d'adressage d'un processus



Pour lancer un programme:

- 1 créer un PCB et une PT
- 2 charger l'exécutable en mémoire
- 3 `PCB.PC` = adresse de main
- 4 `PCB.state` = ready
- 5 ajouter le PCB à la ready queue

Un exécutable = plusieurs sections

- `.text` = les instructions
- `.data` = les variables globales
- `.heap` = le tas
- `.stack` = la pile d'appels

À essayer chez vous:

- `objdump -h ./prog.elf`
- `objdump -d ./prog.elf`

Allocation statique: sections .text et .data

Statique = «qui ne bouge pas pendant l'exécution»

- emplacement fixé avant le début de l'exécution
- taille fixée avant le début de l'exécution

Code source:

```
int i,n,r;

factorial()
{
    i = 1;

    while(n>0)
    {
        i = i*n;
        n = n-1;
    }
    r = i;
}
```

Code exécutable :

```
80483a7:  ....
80483aa:  c7 05 2c 96 04 08 01
80483b1:  00 00 00
80483b4:  eb 20
80483b6:  8b 15 2c 96 04 08
80483bc:  a1 30 96 04 08
80483c1:  0f af c2
80483c4:  a3 2c 96 04 08
80483c9:  a1 30 96 04 08
80483ce:  83 e8 01
80483d1:  a3 30 96 04 08
80483d6:  a1 30 96 04 08
80483db:  85 c0
80483dd:  7f d7
80483df:  a1 2c 96 04 08
80483e4:  a3 34 96 04 08
80483e9:  ...
```

Allocation statique: sections .text et .data

Statique = «qui ne bouge pas pendant l'exécution»

- emplacement fixé avant le début de l'exécution
- taille fixée avant le début de l'exécution

Code source:

```
int i,n,r;

factorial()
{
    i = 1;

    while(n>0)
    {
        i = i*n;
        n = n-1;
    }
    r = i;
}
```

Code exécutable «désassemblé»:

```
80483a7:    ...
80483aa:    movl    $0x1,0x804962c
80483b1:
80483b4:    jmp     0x80483d6
80483b6:    mov     0x804962c,%edx
80483bc:    mov     0x8049630,%eax
80483c1:    imul    %edx,%eax
80483c4:    mov     %eax,0x804962c
80483c9:    mov     0x8049630,%eax
80483ce:    sub     $0x1,%eax
80483d1:    mov     %eax,0x8049630
80483d6:    mov     0x8049630,%eax
80483db:    test    %eax,%eax
80483dd:    jg      0x80483b6
80483df:    mov     0x804962c,%eax
80483e4:    mov     %eax,0x8049634
80483e9:    ...
```


Allocation sur la pile d'exécution (en VO *execution stack*)

Problème: comment faire si la taille ou le nombre de données est **inconnu à l'avance** ?

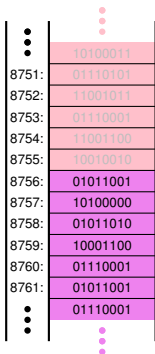
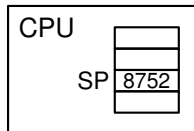
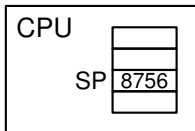
```
int f(int n) {  
    if(n<=1) return 1;  
    int a=f(n-1);  
    int b=f(n-2);  
    return a+b;  
}
```

► **Solution:** une structure de données sans limite, i.e. une **pile**

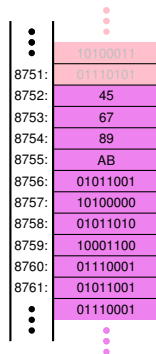
Remarques:

- technique utilisée par 99% des langages de programmation
 - en VO *execution stack*, program stack, control stack, run-time stack, machine stack, call stack, *the stack* = en VF **la pile**
- une activation de fonction = un morceau de la pile
 - variables locales, arguments de fonction, adresses de retour...
- instructions CPU dédiées: PUSH, POP, CALL, RET
 - registre **Stack Pointer** SP : contient adresse du **sommet de pile**

La pile: illustration



PUSH 0x456789AB



Allocation sur la pile: illustration

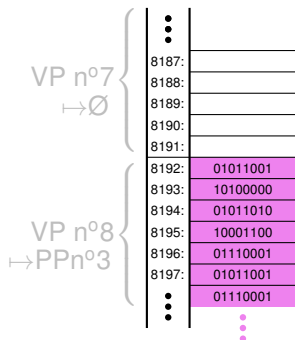
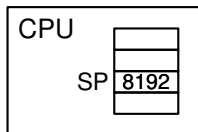
```
int fibo(int n)
{
    if(n<=1)
        return 1;

    int a=f(n-1);
    int b=f(n-2);

    return a+b;
}
```

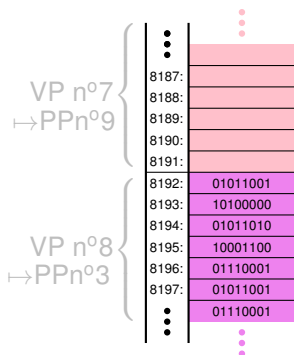
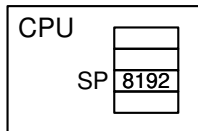
```
8049176 <fibo>:
8049176:  sub    $0x1c,%esp
8049179:  cmpl   $0x1,0x20(%esp)
804917e:  jg     0x8049187
8049180:  mov    $0x1,%eax
8049185:  jmp    0x80491bf
8049187:  mov    0x20(%esp),%eax
804918b:  sub    $0x1,%eax
804918e:  sub    $0xc,%esp
8049191:  push   %eax
8049192:  call   0x8049176
8049197:  add    $0x10,%esp
804919a:  mov    %eax,0xc(%esp)
804919e:  mov    0x20(%esp),%eax
80491a2:  sub    $0x2,%eax
80491a5:  sub    $0xc,%esp
80491a8:  push   %eax
80491a9:  call   0x8049176
80491ae:  add    $0x10,%esp
80491b1:  mov    %eax,0x8(%esp)
80491b5:  mov    0xc(%esp),%edx
80491b9:  mov    0x8(%esp),%eax
80491bd:  add    %edx,%eax
80491bf:  add    $0x1c,%esp
80491c2:  ret
```

Croissance automatique de la pile d'exécution



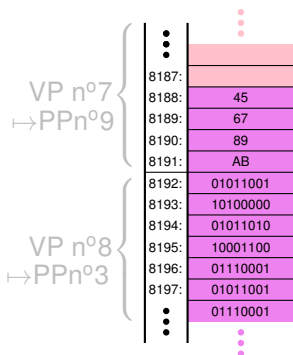
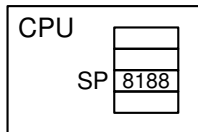
1. application exécute un PUSH
2. MMU cherche à traduire $VA \rightarrow PA$
 - trouve un PTE invalide
 - envoie une IRQ au noyau
3. noyau examine la VA demandée
 - reconnaît un débordement de pile
4. noyau cherche une PP libre
5. noyau «place» la page dans VAS
= met à jour la PT du processus
6. rend la main à l'application
 - instruction PUSH retentée ► OK

Croissance automatique de la pile d'exécution



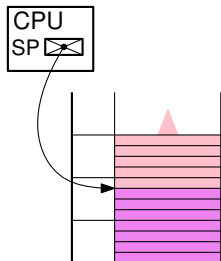
1. application exécute un PUSH
2. MMU cherche à traduire $VA \mapsto PA$
 - trouve un PTE invalide
 - envoie une IRQ au noyau
3. noyau examine la VA demandée
 - reconnaît un **débordement de pile**
4. noyau cherche une PP libre
5. noyau «place» la page dans VAS
= met à jour la PT du processus
6. rend la main à l'application
 - **instruction PUSH retentée** ► OK

Croissance automatique de la pile d'exécution



1. application exécute un PUSH
2. MMU cherche à traduire $VA \rightarrow PA$
 - trouve un PTE invalide
 - envoie une IRQ au noyau
3. noyau examine la VA demandée
 - reconnaît un débordement de pile
4. noyau cherche une PP libre
5. noyau «place» la page dans VAS
= met à jour la PT du processus
6. rend la main à l'application
 - instruction PUSH retentée ► OK

La pile d'exécution: à retenir



Définition

- zone dédiée à l'allocation "dynamique"
 - allocation et désallocation: LIFO
- usage: variables locales, addr retour
 - cf 3IF-architecture, 4IF-compilation
- registre SP pointe sur **sommet de pile**

Avantages

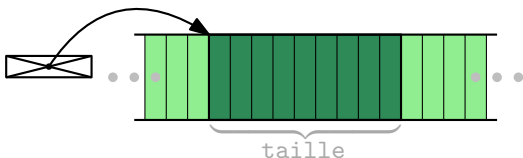
- facile à utiliser et à implémenter
- efficace à l'exécution
 - accès via SP en indirect-register
 - croissance par ajout de page

Inconvénients

- LIFO: inadapté pour certaines structures de données

Allocation dynamique sur le tas (en VO *heap*)

Objectif: permettre allocations et libérations **arbitraires**



Interface utilisateur

- `allouer(taille)`
 - cherche une zone libre et retourne son **adresse** de début (ou renvoie une erreur si incapable de servir la requête)
- `libérer(adresse)`
 - indique au gestionnaire mémoire qu'une zone n'est plus utilisée et qu'elle peut être recyclée

Avantages

- facile à utiliser pour le programmeur
- compatible avec toutes les structures de données

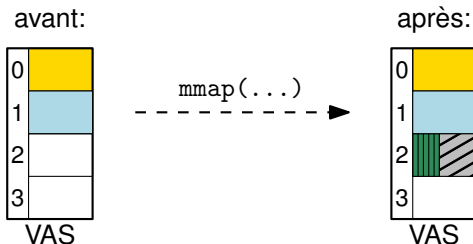
Inconvénients

- implémentation complexe ► performance difficile à maîtriser

Allocation dynamique vs allocation de pages

Question: comment implémenter `allouer()` et `libérer()`

Mauvaise idée: tout déléguer au noyau via `mmap()` et `munmap()`



Inconvénients

- seules tailles disponibles: multiples de 2^p ► espace gâché
- appels système trop fréquents ► mauvaise performance

Solution

- recycler (dans le même processus) les blocs libérés
- besoin de garder la trace des blocs libres et des blocs occupés

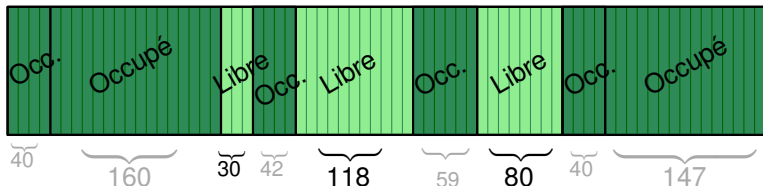
Gestion du tas: formulation de la problématique

Allocation dynamique sur le tas: définition

Un **gestionnaire de mémoire dynamique** (VO memory allocator)

- répond aux **requêtes d'allocation** (resp. de désallocation) émises par l'**application**
- en réservant (resp. en recyclant) des **blocs** de taille variée à l'intérieur d'une grande zone appelée le **tas**

Structure de données: **liste des blocs libres**, en VO *freelist*



Questions: où allouer un bloc de taille 10 ? de 50 ? de 200 ?

Gestion du tas: remarques

Problématique

- à chaque allocation, on veut un bloc **libre** et **assez grand**
- ▶ comment choisir le «meilleur» bloc dans la freelist ?

Règles du jeu

- interdit de «découper» les requêtes d'allocation
 - l'application peut vouloir utiliser le bloc comme un tableau
 - «**allocation contiguë**» ▶ taille allouée \geq taille demandée
- interdit de **réordonner** la séquence des requêtes
 - dépend du flot d'exécution de l'application
- interdit de bouger les zones déjà allouées
 - chaque choix de bloc est définitif

Inconvénients

- trop grand nombre de blocs libres ▶ allocation lente
- blocs libres trop petits ▶ espace inutilisable

Plan

1. Introduction: rappels sur la mémoire virtuelle
2. Allocation dynamique: définition du problème
3. Fragmentation du tas et stratégies d'allocation
4. Techniques d'implémentation et interface utilisateur

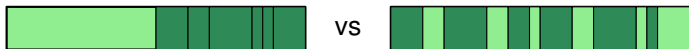
Le problème de la fragmentation

Définition: phénomène de **fragmentation** du tas

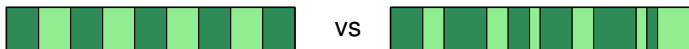
Morcellement progressif de l'espace libre en des blocs trop petits pour satisfaire les requêtes d'allocation de l'application

Les causes de la fragmentation:

- disparité des durées de vie



- disparité des tailles allouées



Problématique: comment **minimiser** la fragmentation du tas ?

Trois pistes: découpage, fusion, choix des blocs

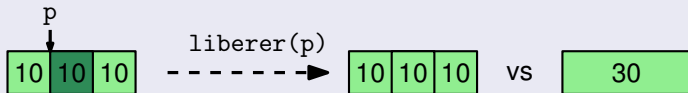
Mécanismes de bas niveau: découpage et fusion

Découpage de blocs libres lors de l'allocation



- ▶ réduit la **fragmentation interne** = espace inutilisé **dans** les blocs
- mais risque de produire des blocs libres (trop) petits

Fusion de blocs libres lors de la déallocation



- ▶ réduit la **fragmentation externe** = blocs trop petits pour être utiles
- mais risque de causer du travail improductif

Stratégies d'allocation: problématique

Comment choisir le «meilleur» bloc dans la freelist ?

Optimal: choisir le bloc qui causera le moins de fragmentation mémoire dans le futur ► impossible à deviner

En pratique: compromis entre fragmentation et performance

- $\text{fragmentation} = \frac{\text{espace total occupé par le tas}}{\text{somme des tailles des blocs alloués}}$
- performance = temps d'exécution de l'algo. de choix de bloc

Note: si aucun bloc libre assez grand ► il faut agrandir le tas

- appel système `mmap()` pour demander des pages au noyau

Stratégies d'allocation: exemples

First-fit : examiner le moins de blocs possible

- parcourir la freelist, et choisir le premier bloc libre de taille suffisante

Next-fit : ne pas retraverser toute la freelist à chaque fois

- variante de First-fit: démarrer le parcours à l'endroit du dernier bloc alloué précédemment

Best-fit : préserver les gros blocs libres

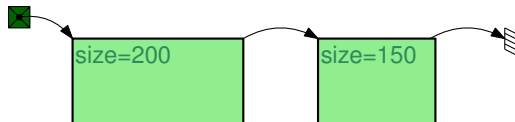
- considérer l'intégralité de la freelist, et choisir le bloc acceptable le plus petit

Worst-fit : éviter la prolifération de blocs minuscules

- considérer l'intégralité de la freelist, et choisir le bloc le plus grand

Recherche dans la freelist: exemple

Supposons la freelist suivante:



Exercice: requêtes d'allocation de 100 octets, puis 200 octets

- stratégies First Fit, Best Fit, Worst Fit

Exercice: requêtes d'allocation de 80 octets, puis 120, puis 120

- stratégies First Fit, Best Fit, Worst Fit

Stratégies d'allocation: commentaires

Mauvaise nouvelle: pas de stratégie universelle

First-fit

- beaucoup de fragmentation en début de liste

Next-fit

- beaucoup de fragmentation partout !

Best-fit

- comment examiner efficacement tous les blocs ?

Worst-fit

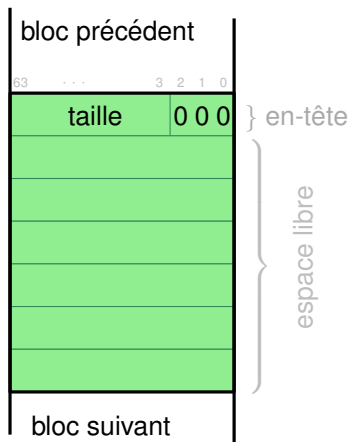
- comment examiner efficacement tous les blocs ?
- énormément de fragmentation

Plan

1. Introduction: rappels sur la mémoire virtuelle
2. Allocation dynamique: définition du problème
3. Fragmentation du tas et stratégies d'allocation
4. Techniques d'implémentation et interface utilisateur
 - Techniques d'implémentation
 - Interface de programmation

Format d'un bloc libre

Idée: indiquer la taille des blocs libres **dans** les blocs

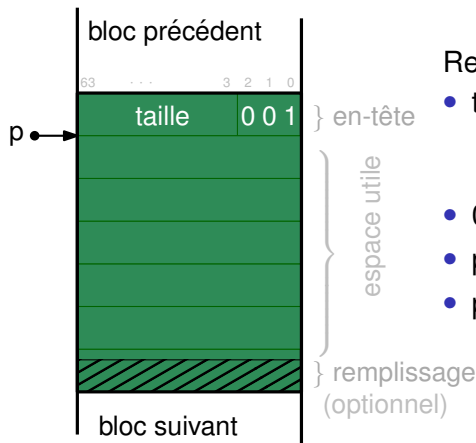


Remarques:

- taille = du bloc entier
 - y compris en-tête
- alignement: toujours multiple de 8
 - ▶ trois derniers bits toujours zéro

Format d'un bloc alloué

Idée: un **drapeau** pour indiquer les blocs alloués



Remarques:

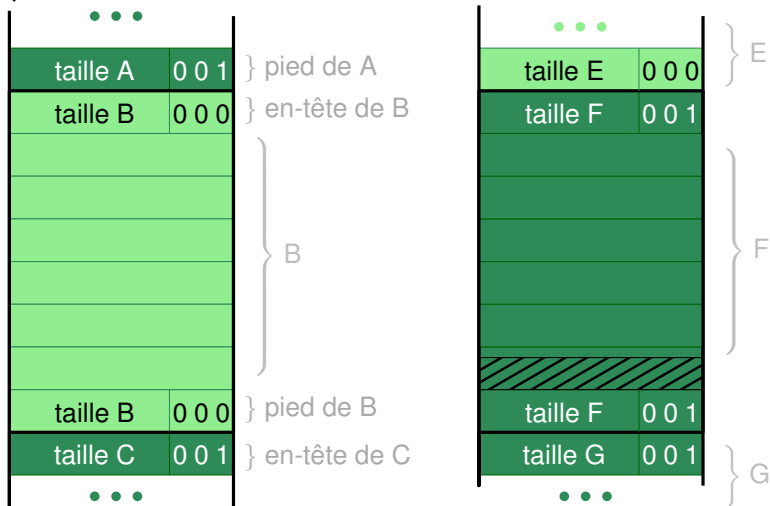
- taille = du bloc entier
 - en-tête + espace utile (+ padding)
 - alignée à 8 octets
- 000 = libre, 001 = alloué
- p = pointeur visible par l'application
- padding = espace perdu

Question: comment localiser le bloc précédent ?

Boundary tags

Idée: ajouter un *footer* à la fin de chaque bloc

► permet de traverser la freelist dans les deux sens

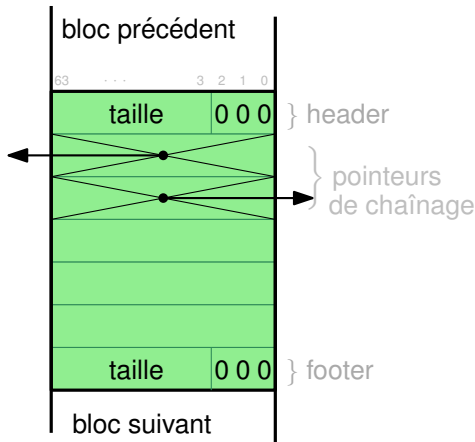


Remarque: impact sur taille de bloc minimale

Chaînage explicite de la freelist

Question: comment accélérer la recherche dans la freelist ?

- Idée 1: **chaîner** uniquement les blocs libres
 - ▶ pas besoin de considérer les blocs alloués
- Idée 2: garder **triée** la freelist dans le «bon» ordre
 - ▶ trouver plus rapidement le bloc recherché



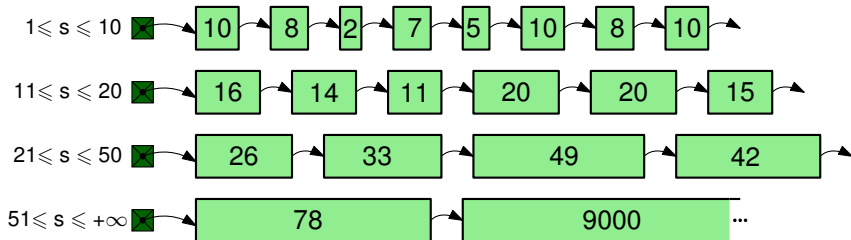
Ordre de tri:

- par adresses croissantes:
 - fusion facile
 - allocation coûteuse
- par tailles croissantes:
 - best-fit efficace
 - fusion coûteuse
- par tailles décroissantes:
 - worst-fit facile
 - fusion coûteuse

La vraie vie: allocateur avec *segregated freelist*

Idée: plusieurs listes chaînées pour les différentes tailles

► approximation de best-fit sans traverser tous les blocs



Remarques

- allocation: si pas trouvé ► découper un bloc plus grand
 - liste triée vs liste non triée
- désallocation: fusion optionnelle ► recyclage des petits blocs
 - cas extrême: tailles exactes, par exemple $s = 8$, $s = 16$, etc.

Exemple: le malloc de Linux

Exemple (glibc) : 128 «paniers» (en VO *bins*)

- 64 premiers paniers: tailles exactes 8, 16, 24, 32, ... 512
- 32 paniers suivants: 513–576, 577–640, ... 2497–2560
- puis 8 espacés de 4kB, 4 espacés de 32kB, deux de 256kB
- au-delà: allocations déléguées à `mmap()`

Rq: chaque sous-liste triée par taille ► best-fit

Free chunks are stored in circular doubly-linked lists, and look like this:

```
chunk-> +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      Size of previous chunk                      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
‘head:’ |                      Size of chunk, in bytes              |P|
mem->   +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      Forward pointer to next chunk in list      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      Back pointer to previous chunk in list     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      Unused space (may be 0 bytes long)         .
.                                                                    .
.                                                                    |
nextchunk-> +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
‘foot:’ |                      Size of chunk, in bytes              |
```

Plan

1. Introduction: rappels sur la mémoire virtuelle
2. Allocation dynamique: définition du problème
3. Fragmentation du tas et stratégies d'allocation
4. Techniques d'implémentation et interface utilisateur
 - Techniques d'implémentation
 - Interface de programmation

Allocation dynamique en C et en C++

Idée: le développeur utilise **explicitement** une API

- **C** ► `void* malloc(size)` et `free(void* ptr)`
- **C++** ► opérateurs `new(size)` et `delete(void* ptr)`

Désallocation **manuelle** = risque d'erreurs de programmation

```
truc *a = malloc(...);
truc *b = a;
b->x = 42;
free(a);
truc *c = malloc(...);
c->x = 37;
printf("b->x: %d\n", b->x);
```

```
truc *a = malloc(...);
truc *b = malloc(...);
b = a ;
```

- **dangling pointer** AKA use-after-free
 - en pratique: assez fréquent et quasi **impossible** à corriger
- fuite de mémoire (memory leak) = blocs jamais désalloués
 - en pratique: seulement gênant si le processus dure **longtemps**
- rappel: `free()` ne rend **pas** la mémoire à l'OS

Allocation dynamique en Java, Python...

Idée 1: si aucun pointeur vers un bloc, alors bloc **inaccessible**

Idée 2: interdire la création de pointeurs arbitraires

► un bloc inaccessible maintenant le restera pour toujours

Définition: Garbage Collection

Allocation dynamique **intégrée** au langage de programmation

- gestionnaire mémoire invoqué **implicitement**
- détection (► libération) automatique des blocs inaccessibles

Inconvénients:

- performance (temps, espace) du Garbage Collector
- incompatible avec programmation «bas-niveau» e.g. C/C++

Avantages:

- (parfois) gain de performances à l'exécution
- **plus simple à programmer: pas de `free()`**

Plan

1. Introduction: rappels sur la mémoire virtuelle
2. Allocation dynamique: définition du problème
3. Fragmentation du tas et stratégies d'allocation
4. Techniques d'implémentation et interface utilisateur

À retenir: allocation dynamique de mémoire

Allocation sur la pile = variables locales

- simple et rapide car fragmentation impossible
- LIFO: limité à certains usages

Allocation dynamique = Allocation sur le tas = `malloc()`

- pas de restrictions sur les structures de données
- risque de **fragmentation** (interne et/ou externe)
- ▶ algorithmes complexes pour allouer/désallouer
- **pas de solution miracle universelle**

Allocation manuelle en C/C++

- erreurs de programmation fréquentes
- ▶ pas toujours utile de scrupuleusement désallouer toute votre mémoire !