LINFO1252 - LSINC1252
Systèmes Informatiques

Leçon 3 : Gestion de la mémoire

Pr. Etienne Rivière
etienne.riviere@uclouvain.be

Parties du syllabus couvertes

- Le langage C
- Types de données
- Déclarations
- Unions et énumérations
- Organisation de la mémoire
- Gestion de mémoire dynamique
- Compléments de C
  - Pointeurs
  - De grands programmes en C
  - Traitement des erreurs

couvertes la semaine passée

couvertes aujourd'hui

à lire par vous même

Objectifs de ce cours

- Décrire l'organisation de la mémoire en C
- Apprendre comment utiliser et mettre en œuvre les fonctions de gestion dynamique de la mémoire

Organisation de la mémoire

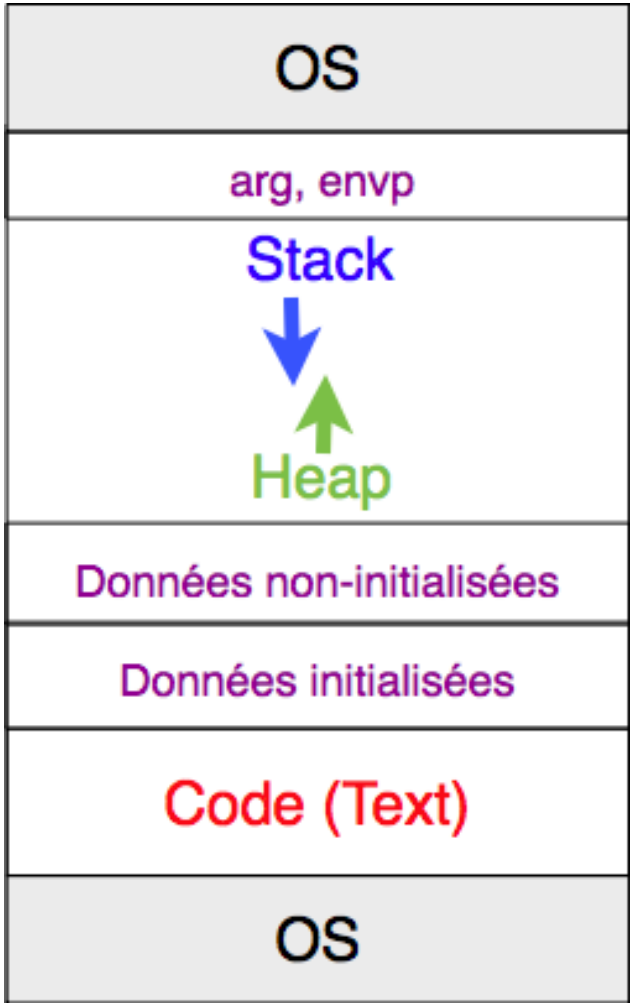
Introduction

- L'exécution d'un programme sous Unix créée
  - Un nouveau processus (abstraction d'un fil d'exécution d'instruction sur un processeur)
  - Un nouvel espace mémoire spécifique pour ce processus (isolé de celui des autres processus)
- L'espace mémoire suit une organisation pré-définie en 6 zones ou *segments*
- Certaines de ces zones sont initialisées en lisant le contenu d'un fichier programme depuis le disque
  - Instructions du programme (générées par le compilateur)
  - Données statiques, initialisées et non-initialisées

Organisation d'un programme Linux en mémoire : 6 zones

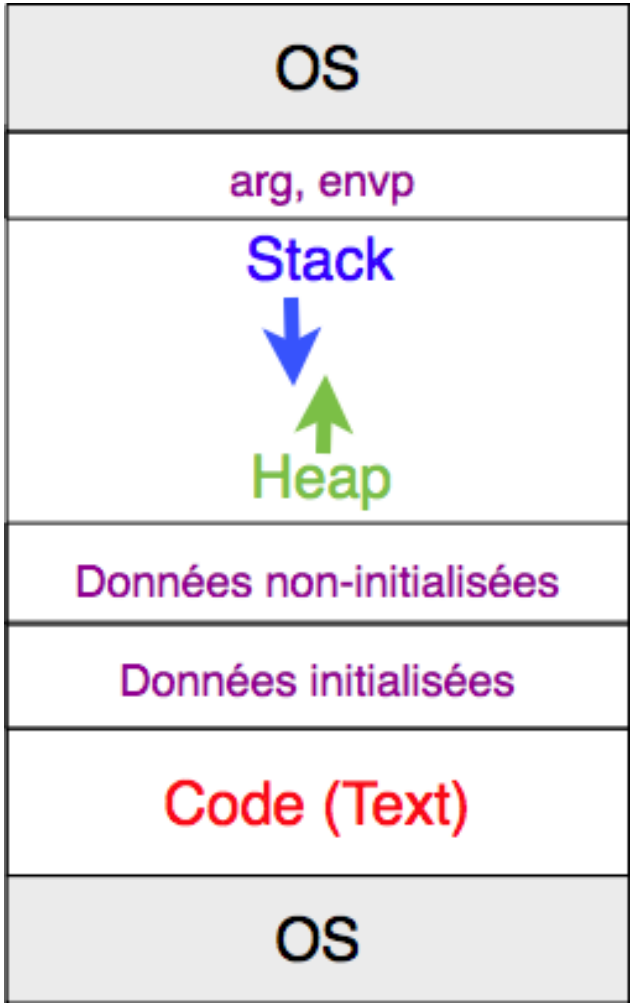
- Les parties grisées correspondent à des espaces mémoire réservés au système d'exploitation (OS)

- Segment text
  - En partie basse de la mémoire
  - Contient les instructions à exécuter par le processeur
  - Accès en lecture seule (pourquoi ?)
    - Une opération d'écriture déclenchera une interruption (trap) par le processeur
    - Le SE reprend le contrôle et termine le processus



Organisation d'un programme Linux en mémoire : 6 zones

- Segment de données statiques
  - Séparée en deux parties
- Données initialisées
  - Variables globales explicitement initialisées par le programme
  - Variables statiques
- Données non-initialisées
  - Mises à zéro par le compilateur



Données (non)initialisées

```c
#define MSG_LEN 10
int g;   // initialisé par le compilateur
int g_init=1252;
const int un=1;
int tab[3]={1,2,3};
int array[10000];
char cours[]="SINF1252";
char msg[MSG_LEN]; // initialisé par le compilateur

int main(int argc, char *argv[]) {
  int i;
  printf("g est à l'adresse %p et initialisée à %d\n",&g,g);
  printf("msg est à l'adresse %p contient les caractères :",msg);
  for(i=0;i<MSG_LEN;i++)
    printf(" %x",msg[i]);
  printf("\n");
  printf("Cours est à l'adresse %p et contient : %s\n",&cours,cours);
  return(EXIT_SUCCESS);
}
```

```
g est à l'adresse 0x60aeac et initialisée à 0
msg est à l'adresse 0x60aea0 contient les caractères : 0 0 0 0 0 0 0 0 0 0
Cours est à l'adresse 0x601220 et contient : SINF1252
```

⚠ Initialisation des variables (I)

- Contrairement aux variables globales non initialisées, des variables locales non initialisées par le programmeur ne sont pas initialisées à 0 par le compilateur
- Exemple :
  - `read()` lit les valeurs d'un tableau qui ne sont pas initialisées
  - `init()` positionne des valeurs dans un tableau qui n'est pas renvoyé avec return

```
#define ARRAY_SIZE 1000

// initialise un tableau local
void init(void) {
  long a[ARRAY_SIZE];
  for(int i=0;i<ARRAY_SIZE;i++) {
    a[i]=i;
  }
}

// retourne la somme des éléments
// d'un tableau local
long read(void) {
  long b[ARRAY_SIZE];
  long sum=0;
  for(int i=0;i<ARRAY_SIZE;i++) {
    sum+=b[i];
  }
  return sum;
}
```

⚠ Initialisation des variables (2)

- Exécution :

```
printf("Résultat de read() avant init(): %ld\n",read());
init();
printf("Résultat de read() après init() : %ld\n",read());
```

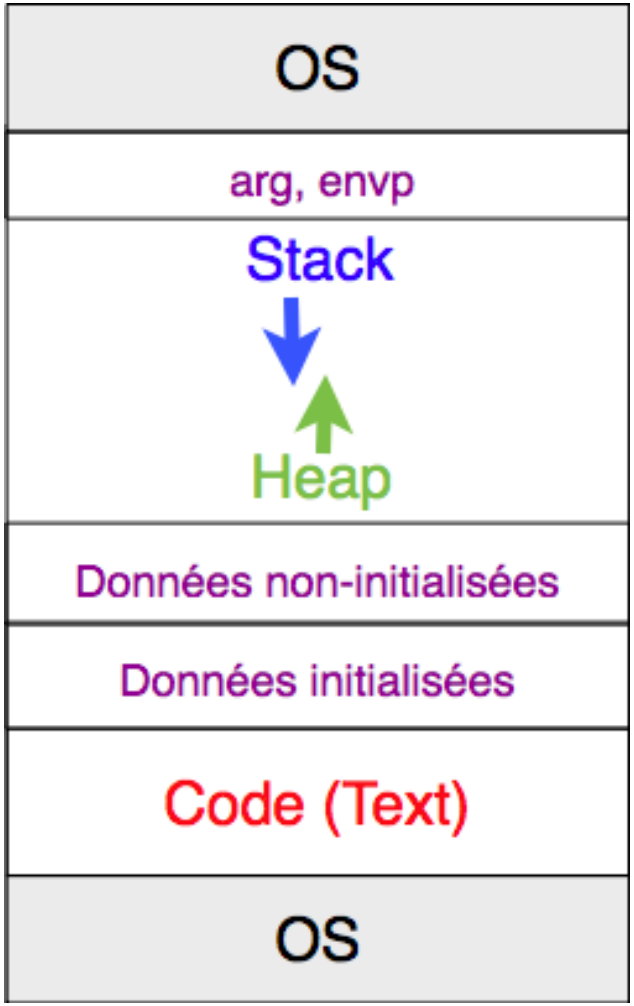
```
Résultat de read() avant init(): 7392321044987950589
Résultat de read() après init() : 499500
```

- $\sum_{i=1}^{1000} i = 499500$
- Pouvez vous expliquer ?

```
#define ARRAY_SIZE 1000

// initialise un tableau local
void init(void) {
  long a[ARRAY_SIZE];
  for(int i=0;i<ARRAY_SIZE;i++) {
    a[i]=i;
  }
}

// retourne la somme des éléments
// d'un tableau local
long read(void) {
  long b[ARRAY_SIZE];
  long sum=0;
  for(int i=0;i<ARRAY_SIZE;i++) {
    sum+=b[i];
  }
  return sum;
}
```

Le tas (heap)

- "Démarre" après le segment de données non initialisées
- Un programme peut y réserver des zones mémoire pour y stocker de l'information
  - La réservation renvoie un pointeur vers le début de la zone réservée
  - Lorsque la zone n'est plus nécessaire, il faut la libérer en fournissant le pointeur d'origine
- Gestion manuelle du heap possible mais compliquée : on préfère l'utilisation des fonctions de la librairie standard `malloc` et `free`



```
NAME      top
       malloc, free, calloc, realloc, reallocarray - allocate and free
       dynamic memory
SYNOPSIS      top
       #include <stdlib.h>

       void *malloc(size_t size);
       void free(void *ptr);
       void *calloc(size_t nmemb, size_t size);
       void *realloc(void *ptr, size_t size);
       void *reallocarray(void *ptr, size_t nmemb, size_t size);
(…)
DESCRIPTION      top
       The malloc() function allocates size bytes and returns a pointer to
       the allocated memory.  The memory is not initialized.  If size is 0,
       then malloc() returns either NULL, or a unique pointer value that can
       later be successfully passed to free().
       The free() function frees the memory space pointed to by ptr, which
       must have been returned by a previous call to malloc(), calloc(), or
       realloc().  Otherwise, or if free(ptr) has already been called
       before, undefined behavior occurs.  If ptr is NULL, no operation is
       performed.
```

- `malloc` prend un argument de type `size_t` et retourne un pointeur vers la zone allouée
  - Toujours utiliser `sizeof(type)` pour calculer la taille, car celle-ci peut différer d'un système à un autre !
  - Le pointeur retourné est un `void *` qui doit être "casté" :
    - `int * mytab = (int *) malloc(1000*sizeof(int));`

```
NAME      top
       malloc, free, calloc, realloc, reallocarray - allocate and free
       dynamic memory
SYNOPSIS      top
       #include <stdlib.h>

       void *malloc(size_t size);
       void free(void *ptr);
       void *calloc(size_t nmemb, size_t size);
       void *realloc(void *ptr, size_t size);
       void *reallocarray(void *ptr, size_t nmemb, size_t size);
(…)
DESCRIPTION      top
       The malloc() function allocates size bytes and returns a pointer to
       the allocated memory.  The memory is not initialized.  If size is 0,
       then malloc() returns either NULL, or a unique pointer value that can
       later be successfully passed to free().
       The free() function frees the memory space pointed to by ptr, which
       must have been returned by a previous call to malloc(), calloc(), or
       realloc().  Otherwise, or if free(ptr) has already been called
       before, undefined behavior occurs.  If ptr is NULL, no operation is
       performed.
```

- `free` prend un pointeur en argument et ne renvoie rien
  - si ce pointeur n'est pas un pointeur précédemment retourné par un appel à malloc, le comportement est indéfini (et rarement bénéfique …)
  - si `free(ptr)` a déjà été appelé, idem !
  - `free(NULL)` n'a pas d'effet

Exemple (1)

```c
int size=20;

char * string;
printf("Valeur du pointeur string avant malloc : %p\n",string);
string=(char *) malloc((size+1)*sizeof(char));
if(string==NULL)
  error("malloc(string)");

printf("Valeur du pointeur string après malloc : %p\n",string);
int *vector;
vector=(int *)malloc(size*sizeof(int));
if(vector==NULL)
  error("malloc(vector)");

free(string);
printf("Valeur du pointeur string après free   : %p\n",string);
string=NULL;
free(vector);
vector=NULL;
```

⚠ un octet supplémentaire pour stocker '\0' !

```
Valeur du pointeur string avant malloc : 0x7fff5fbfe1d8
Valeur du pointeur string après malloc : 0x100100080
Valeur du pointeur string après free   : 0x100100080
```

Exemple (2)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

typedef struct fraction {
  int num;
  int den;
} Fraction;

void error(char *msg) {
  fprintf(stderr,"Erreur :%s\n",msg);
  exit(EXIT_FAILURE);
}

int main(int argc, char *argv[]) {
  int size=1000;

  Fraction * fract_vect;
  fract_vect=(Fraction *) malloc(size*sizeof(Fraction));
  if(fract_vect==NULL)
    error("malloc(fract_vect)");

  free(fract_vect);
  fract_vect=NULL;
}
```

Bonnes pratiques

- Utilisation de conversion explicite avec le cast de la valeur de retour du `malloc`
  - Meilleure lisibilité
  - Le compilateur ne génère pas de warning (conversion de type implicite) !
- Toujours tester le pointeur retourné par `malloc` : si il est `NULL` c'est qu'il n'y a plus de mémoire disponible
  - Il est préférable de terminer le programme 'proprement' que de tenter d'écrire à l'adresse 0 (segmentation fault assuré)
- `free(ptr)` libère la zone réservée pointée par `ptr` mais ne remet pas `ptr` à zéro
  - Et ne le pourrait pas car on fournit l'adresse contenue dans `ptr` à `free(ptr)` par valeur ; il faudrait utiliser un pointeur vers un pointeur
  - Toujours positionner un pointeur à `NULL` après le `free` correspondant

Exemple : la pile

```c
// affiche le contenu de la pile
void display()
{ struct node_t *t;
  t = stack;
  while(t!=NULL) {
    if(t->data!=NULL) {
      printf("Item at addr %p  :  Fraction %d/%d   Next %p\n",
             t,t->data->num,t->data->den,t->next);
    }
    else {
      printf("Bas du stack %p\n",t);
    }
    t=t->next; }}

// exemple
int main(int argc, char *argv[]) {

  struct fraction_t demi={1,2}; struct fraction_t tiers={1,3};
  struct fraction_t quart={1,4}; struct fraction_t zero={0,1};

  // initialisation
  stack = (struct node_t *)malloc(sizeof(struct node_t));
  stack->next=NULL; stack->data=NULL;

  display();
  push(&zero);
  display();
  push(&demi); push(&tiers); push(&quart);
  display();

  struct fraction_t *f=pop();
  if(f!=NULL)
    printf("Popped : %d/%d\n",f->num,f->den);

  return(EXIT_SUCCESS);
}
```

```c
typedef struct node_t
{
  struct fraction_t *data;
  struct node_t *next;
} node;

struct node_t *stack; // sommet de la pile

// ajoute un élément au sommet de la pile
void push(struct fraction_t *f)
{
  struct node_t *n;
  n=(struct node_t *)malloc(sizeof(struct node_t));
  if(n==NULL)
    exit(EXIT_FAILURE);
  n->data = f;
  n->next = stack;
  stack = n;
}
// retire l'élément au sommet de la pile
struct fraction_t * pop()
{
  if(stack==NULL)
    return NULL;
  // else
  struct fraction_t *r;
  struct node_t *removed=stack;
  r=stack->data;
  stack=stack->next;
  free(removed);
  return (r);
}
```

```
Bas du stack 0x100100080
Item at addr 0x100100090  :  Fraction 0/1   Next 0x100100080
Bas du stack 0x100100080
Item at addr 0x1001000c0  :  Fraction 1/4   Next 0x1001000b0
Item at addr 0x1001000b0  :  Fraction 1/3   Next 0x1001000a0
Item at addr 0x1001000a0  :  Fraction 1/2   Next 0x100100090
Item at addr 0x100100090  :  Fraction 0/1   Next 0x100100080
Bas du stack 0x100100080
Popped : 1/4
```

Fuites mémoire (*memory leaks*)

```c
#define LEN 1024
int main(int argc, char *argv[]) {

  char *str=(char *) malloc(sizeof(char)*LEN);
  for(int i=0;i<LEN-1;i++) {
    *(str+i)='A';
  }
  *(str+LEN)='\0';  // fin de chaîne
  return(EXIT_SUCCESS);
}
```

- Ce code termine sans erreur, et les ressources mémoires sont libérés par le SE
  - Peut on ignorer `free()` dans ce cas ?
  - Une mauvaise habitude …
  - Problème de fuite mémoire : si le programme fonctionne en continu, la mémoire finira par saturer

Utilisation de `valgrind`

```
vagrant@precise32:/tmp/test_c$ gcc -std=c99 -o leaky leaky.c
vagrant@precise32:/tmp/test_c$ valgrind ./leaky
==12613== Memcheck, a memory error detector
==12613== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==12613== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==12613== Command: ./leaky
==12613==
==12613== Invalid write of size 1
==12613==    at 0x804842A: main (in /tmp/test_c/leaky)
==12613==  Address 0x41e8428 is 0 bytes after a block of size 1,024 alloc'd
==12613==    at 0x402BE68: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-
linux.so)
==12613==    by 0x80483F8: main (in /tmp/test_c/leaky)
==12613==
==12613==
==12613== HEAP SUMMARY:
==12613==     in use at exit: 1,024 bytes in 1 blocks
==12613==   total heap usage: 1 allocs, 0 frees, 1,024 bytes allocated
==12613==
==12613== LEAK SUMMARY:
==12613==    definitely lost: 1,024 bytes in 1 blocks
==12613==    indirectly lost: 0 bytes in 0 blocks
==12613==      possibly lost: 0 bytes in 0 blocks
==12613==    still reachable: 0 bytes in 0 blocks
==12613==         suppressed: 0 bytes in 0 blocks
==12613== Rerun with --leak-check=full to see details of leaked memory
==12613==
==12613== For counts of detected and suppressed errors, rerun with: -v
==12613== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

```c
#define LEN 1024
int main(int argc, char *argv[]) {

  char *str=(char *) malloc(sizeof(char)*LEN);
  for(int i=0;i<LEN-1;i++) {
    *(str+i)='A';
  }
  *(str+LEN)='\0';  // fin de chaîne
  return(EXIT_SUCCESS);
}
```

- Plus de détails dans la section Outils du Syllabus

Variantes de malloc

```
SYNOPSIS       top
       void *calloc(size_t nmemb, size_t size);
(…)
DESCRIPTION       top
       The calloc() function allocates memory for an array of nmemb elements
       of size bytes each and returns a pointer to the allocated memory.
       The memory is set to zero.  If nmemb or size is 0, then calloc()
       returns either NULL, or a unique pointer value that can later be
       successfully passed to free().  If the multiplication of nmemb and
       size would result in integer overflow, then calloc() returns an
       error.  By contrast, an integer overflow would not be detected in the
       following call to malloc(), with the result that an incorrectly sized
       block of memory would be allocated:

           malloc(nmemb * size);
```

- `calloc` initialise une zone mémoire de `size*nmemb` octets
  - vérification d'un possible dépassement de la valeur maximale permise pour un `size_t` (après la multiplication)
  - contrairement à `malloc`, la zone réservée est mise à 0
- `malloc` est plus rapide que `calloc` (la mise à zéro est inutile si la zone est immédiatement remplie de données)

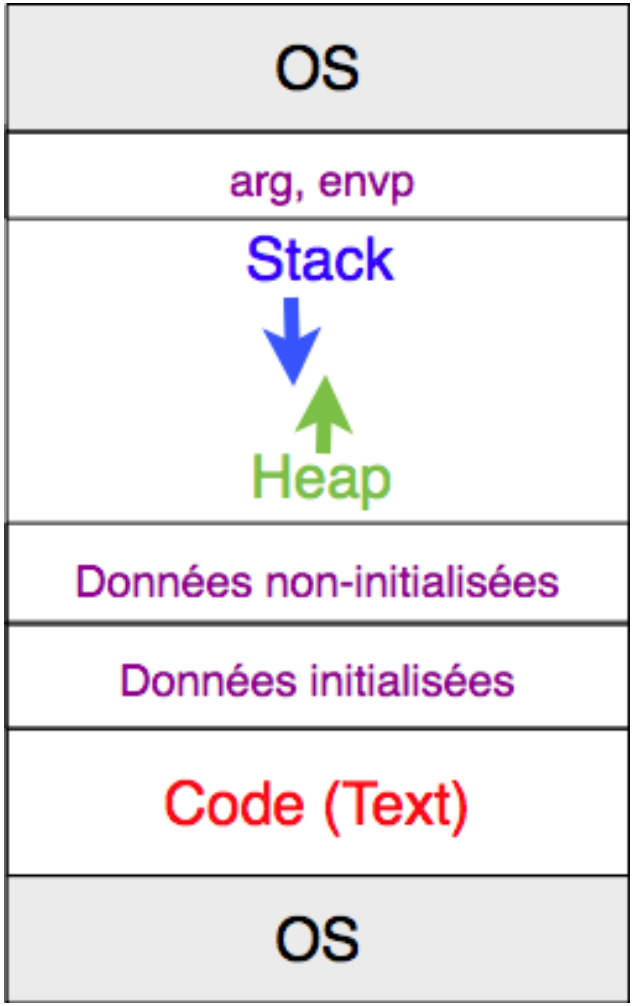
Arguments et variables d'environnement

- Une zone dans le 'haut' de la mémoire contient des données contextuelles préparées par le SE et accessibles en lecture seule par le programme
- Arguments de la ligne de commande (contenu pointé par `**argv`)
- Variables d'environnement
  - Spécifique au système hôte et à l'utilisateur
  - Exemples: `HOSTNAME`, `SHELL`, `USER`, `HOME`, `PATH`, etc.
  - `getenv`, `unsetenv`, `setenv` : fonctions de la librairie standard permettant de manipuler les variables d'environnement

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

// affiche la valeur de la variable d'environnement var
void print_var(char *var) {
  char *val=getenv(var);
  if(val!=NULL)
    printf("La variable %s a la valeur : %s\n",var,val);
  else
    printf("La variable %s n'a pas été assignée\n",var);
}

int main(int argc, char *argv[]) {

  char *old_path=getenv("PATH");

  print_var("PATH");

  if(unsetenv("PATH")!=0) {
    fprintf(stderr,"Erreur unsetenv\n");
    exit(EXIT_FAILURE);
  }

  print_var("PATH");

  if(setenv("PATH",old_path,1)!=0) {
    fprintf(stderr,"Erreur setenv\n");
    exit(EXIT_FAILURE);
  }

  print_var("PATH");

  return(EXIT_SUCCESS);
}
```

La pile (stack)

- "Démarre" depuis le haut de l'espace mémoire, après les informations d'environnement
- Stocke les variables locales, et permet l'appel de fonctions :
  - Stockage des paramètres d'appel
  - Récupération de la valeur de retour
- Comme son nom l'indique, elle est gérée comme une pile LIFO (Last-In-First-Out — *dernier arrivé, dernier parti*)
  - Les adresses décroissent quand la pile croît



```c
// retourne i*j
int times(int i, int j) {
  int m;
  m=i*j;
  printf("\t[times(%d,%d)] : return(%d)\n",i,j,m);
  return m;
}
// calcul récursif de factorielle
// n>0
int fact(int n) {
  printf("[fact(%d)]: Valeur de n:%d, adresse: %p\n",n,n,&n);
  int f;
  if(n==1) {
    printf("[fact(%d)]: return(1)\n",n);
    return(n);
  }
  printf("[fact(%d)]: appel à fact(%d)\n",n,n-1);
  f=fact(n-1);
  printf("[fact(%d)]: calcul de times(%d,%d)\n",n,n,f);
  f=times(n,f);
  printf("[fact(%d)]: return(%d)\n",n,f);
  return(f);
}

void compute() {
  int nombre=3;
  int f;
  printf("La fonction fact est à l'adresse : %p\n",fact);
  printf("La fonction times est à l'adresse : %p\n",times);
  printf("La variable nombre vaut %d et est à l'adresse
%p\n",nombre,&nombre);
  f=fact(nombre);
  printf("La factorielle de %d vaut %d\n",nombre,f);
}
```

`fact` et `times` dans le segment text (commence à `0x100000000` sous Linux)

`nombre` en haut de la mémoire sur le stack (le haut de l'espace utilisateur est `0x7FFFFFFF` sous Linux)

Les paramètres d'appel `n` des appels à `fact()` sont situés plus haut dans la pile (et donc avec adresses descendantes)

```
La fonction fact est à l'adresse : 0x100000a0f
La fonction times est à l'adresse : 0x1000009d8
La variable nombre vaut 3 et est à l'adresse 0x7fff5fbfe1ac
[fact(3)]: Valeur de n:3, adresse: 0x7fff5fbfe17c
[fact(3)]: appel à fact(2)
[fact(2)]: Valeur de n:2, adresse: 0x7fff5fbfe14c
[fact(2)]: appel à fact(1)
[fact(1)]: Valeur de n:1, adresse: 0x7fff5fbfe11c
[fact(1)]: return(1)
[fact(2)]: calcul de times(2,1)
      [times(2,1)] : return(2)
[fact(2)]: return(2)
[fact(3)]: calcul de times(3,2)
      [times(3,2)] : return(6)
[fact(3)]: return(6)
La factorielle de 3 vaut 6
```

Passage de paramètres

- Les paramètres d'appel d'une fonction sont passés par valeur en utilisant la pile
  - Ajout (`push`) des valeurs des paramètres dans l'ordre, puis ajout de l'adresse de retour
  - Appel à l'instruction jump (`jmp`) vers l'adresse de la première instruction de la fonction
- Si le paramètre est un `int` : la copie est OK
- Mais si le paramètre est une structure de donnée de grande taille ?
  - Chaque appel va créer une copie …

Illustration

- Deux versions d'une fonction additionnant un champ d'une grande structure (1.000.001 octets)
- `sum` : copie intégrale du contenu des deux structures fournies en argument
- `sum_ptr` : copie uniquement de leur adresse
- 100s de microsecondes vs. < 1 microseconde à l'exécution !

```c
#define MILLION 1000000

struct large_t {
  int i;
  char str[MILLION];
};

int sum(struct large_t s1, struct large_t s2) {
  return (s1.i+s2.i);
}

int sumptr(struct large_t *s1, struct large_t *s2) {
  return (s1->i+s2->i);
}

int main(int argc, char *argv[]) {
  struct timeval tvStart, tvEnd;
  int err;
  int n;
  struct large_t one={1,"one"};
  struct large_t two={1,"two"};

  n=sum(one,two);
  n=sumptr(&one,&two);
```

⚠ Initialisation des variables (3) Explication du “mystère”

- Exécution :

```
printf("Résultat de read() avant init(): %ld\n",read());
init();
printf("Résultat de read() après init() : %ld\n",read());
```

```
Résultat de read() avant init(): 7392321044987950589
Résultat de read() après init() : 499500
```

- Le tableau `a`, variable locale de `init()` a été initialisé sur la pile, et que `read()` déclare un tableau `b` de même taille ; comme `init()` a terminé son exécution le sommet de pile lors de l’appel à `read()` est à la même adresse que lors de l’appel à `init()`

```
#define ARRAY_SIZE 1000

// initialise un tableau local
void init(void) {
  long a[ARRAY_SIZE];
  for(int i=0;i<ARRAY_SIZE;i++) {
    a[i]=i;
  }
}

// retourne la somme des éléments
// d'un tableau local
long read(void) {
  long b[ARRAY_SIZE];
  long sum=0;
  for(int i=0;i<ARRAY_SIZE;i++) {
    sum+=b[i];
  }
  return sum;
}
```

⚠ Allocation de tableau sur pile

- Version récente de C (C99) permettent d'allouer un tableau sur la pile dynamiquement (lors de l'exécution) ; la taille n'a pas à être connue à l'avance
- Utile dans certains cas (tableaux temporaires)
- Mais attention à ne pas les utiliser comme valeur de retour !
  - Les données locales sur la pile sont en effet libérées à la fin de l'exécution de la fonction

Illustration

```c
#include <string.h>

char *duplicate(char * str) {
  int i;
  size_t len=strlen(str);
  char *ptr=(char *)malloc(sizeof(char)*(len+1));
  if(ptr!=NULL) {
    for(i=0;i<len+1;i++) {
     *(ptr+i)=*(str+i);
      }
  }
  return ptr;
}
```

VS

```c
char *duplicate2(char * str) {
  int i;
  size_t len=strlen(str);
  char str2[len+1];
  for(i=0;i<len+1;i++) {
    str2[i]=*(str+i);
  }
  return str2;
}
```

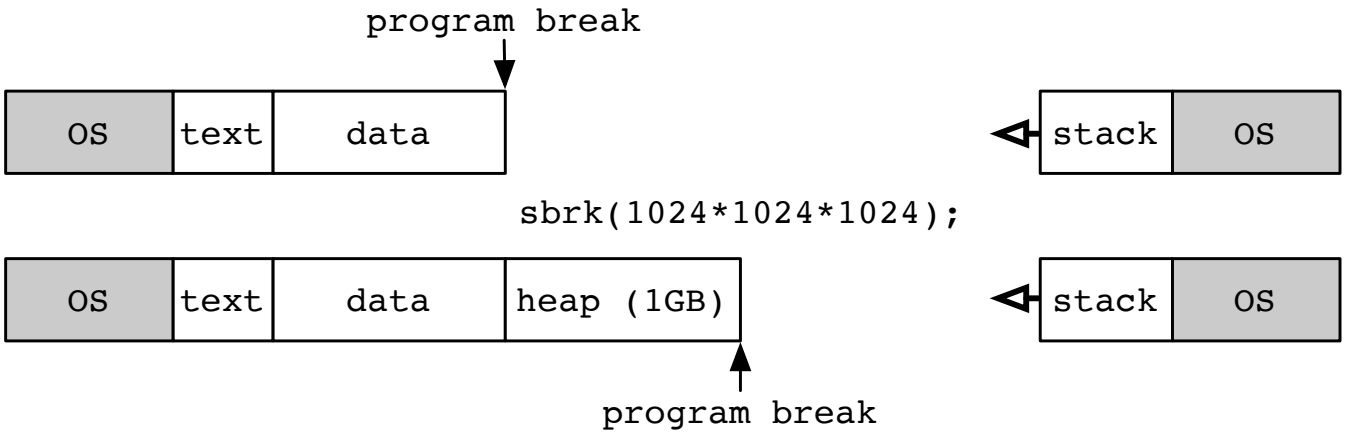
incorrect !

Gestion de la mémoire dynamique

Récapitulatif

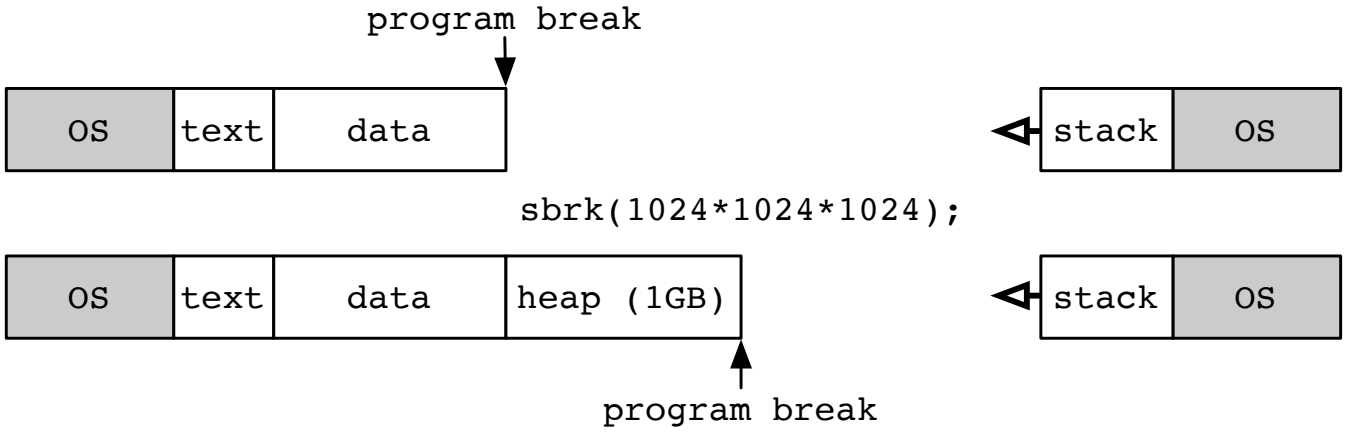
- Nous avons vu que la mémoire du tas (heap) est réservée et libérée en utilisant les fonctions de la librairie standard `malloc()` et `free()`
- Ces fonctions ne sont pas mises en œuvre dans le noyau du système d'exploitation : ce ne sont pas des appels systèmes
  - Ce sont des fonctions de la librairie standard
  - Néanmoins, elle peuvent utiliser des appels systèmes lorsque c'est nécessaire, pour étendre la taille de la zone mémoire dédiée au heap

Gestion du heap



- Le heap démarre après les segments `data`
  - Première adresse après la fin du segment data
- Le *program break* détermine la fin du segment
- Initialement, *program break* == début du heap
  - Le heap est donc initialement de taille 0

Gestion du heap




```c
#include <unistd.h>
int brk(void *addr);
void *sbrk(intptr_t increment);
```

- Appels système `brk` et `sbrk`
  - `brk` : positionne program break a une adresse
  - `sbrk` : décale le program break d'un incrément et retourne la valeur du program break résultat
    - `sbrk(0)` retourne la valeur actuelle

Limitations mémoire

- `brk` et `sbrk` peuvent générer une erreur si l'espace mémoire demandé pour le heap amène à un dépassement de la mémoire maximale autorisée pour le processus
  - Erreur `ENOMEM` : le processus sera arrêté par le SE
- Cette valeur dépend du système et des contraintes de l'utilisateur
  - Mises en place par l'administrateur du système
- On peut consulter cette limite en utilisant l'utilitaire ulimit (`ulimit -a`)

BRK

NAME
brk, sbrk - change data segment size
SYNOPSIS
#include <unistd.h>
int brk(void *_addr_);
void *sbrk(intptr_t _increment_);

DESCRIPTION
brk() and **sbrk**() change the location of the *program break*, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.
brk() sets the end of the data segment to the value specified by *addr*, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum data size (see **setrlimit**(2)).
sbrk() increments the program's data space by *increment* bytes. Calling **sbrk**() with an *increment* of 0 can be used to find the current location of the program break.

RETURN VALUE
On success, **brk**() returns zero. On error, -1 is returned, and *errno* is set to **ENOMEM**. (But see *Linux Notes* below.)
On success, **sbrk**() returns the previous program break. (If the break was increased, then this value is a pointer to the start of the newly allocated memory). On error, *(void *) -1* is returned, and *errno* is set to **ENOMEM**.

malloc et free

- En pratique, on n'utilise jamais directement les appels systèmes `brk` et `sbrk`

- La fonction de la librairie standard `malloc()` se charge d'utiliser ces appels systèmes
  - C'est toujours le cas lors de la première invocation !
  - Ensuite, seulement lorsque la mémoire disponible dans le heap ne permet pas de répondre à une demande d'allocation

- `free()` peut aussi décider de réduire la taille du segment

- Nécessité d'un algorithme de gestion de mémoire dynamique

Algorithme de gestion de mémoire dynamique

- Contraintes
  - Conserver de l'information sur les blocs alloués et libérés : méta-données
  - Ces méta-données ne peuvent être stockées que dans le segment heap lui-même
  - Il faut donc intercaler les méta-données, permettant le suivi des allocations et des libérations, avec les blocs de données eux-même

Alignement

- Les zones mémoire allouées par malloc() sont alignées sur un *facteur d'alignement*
  - Nombre d'octets entiers, généralement multiple de 2
  - Sous Linux actuellement : alignement sur 16 octets (128 bits)
- Une demande de mémoire retourne le multiple immédiatement supérieur du nombre d'octets demandé
  - Une demande de 17 octets réservera 32 octets
  - ➡ Principe de *padding* (remplissage)
- Intérêts
  - On peut faire des hypothèses sur les adresses retournées
    - Bits de poids faible à zéro
  - Ceci facilite la mise en œuvre efficace et énonome en mémoire des algorithmes de gestion de la mémoire dynamique

Exemple sous Linux

```
char *a, *b, *c;

a = (char *) malloc(sizeof(char)*1);
b = (char *) malloc(sizeof(char)*9);
c = (char *) malloc(sizeof(char)*1);

printf("Adresse de a : %p.\n",a);
printf("Adresse de b : %p.\n",b);
printf("Adresse de c : %p.\n",c);
```

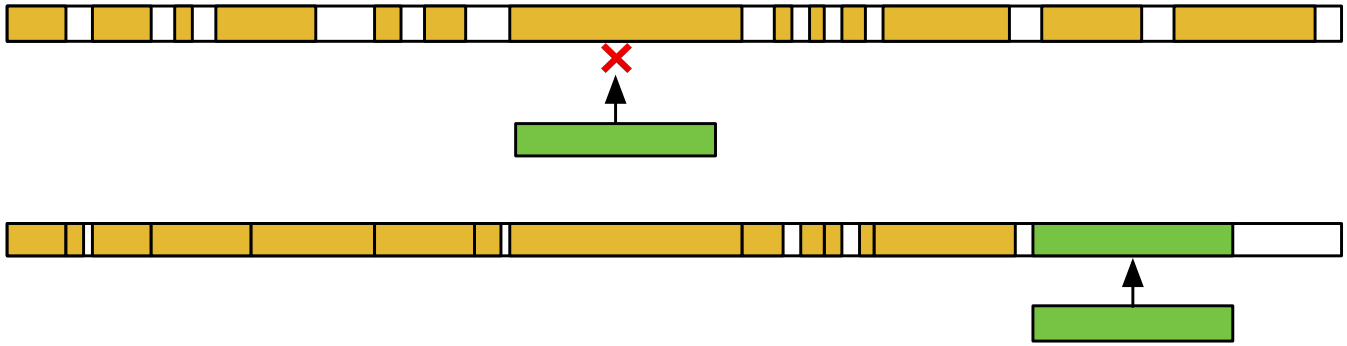
```
Adresse de a : 0x8e29008.
Adresse de b : 0x8e29018.
Adresse de c : 0x8e29028.
```

Objectifs

- Trois critères principaux pour la qualité d'un algorithme de gestion de la mémoire dynamique :
  - Temps d'exécution des fonctions malloc() et free()
    - Est-il faible ?
    - Est-il stable ?
    - Allocation/libération mémoire sur le chemin critique dans de nombreuses applications
  - Utilisation efficace de la mémoire disponible : faible fragmentation
  - Bonne propriétés de localité spatiale
- Ces objectifs peuvent être contradictoires !

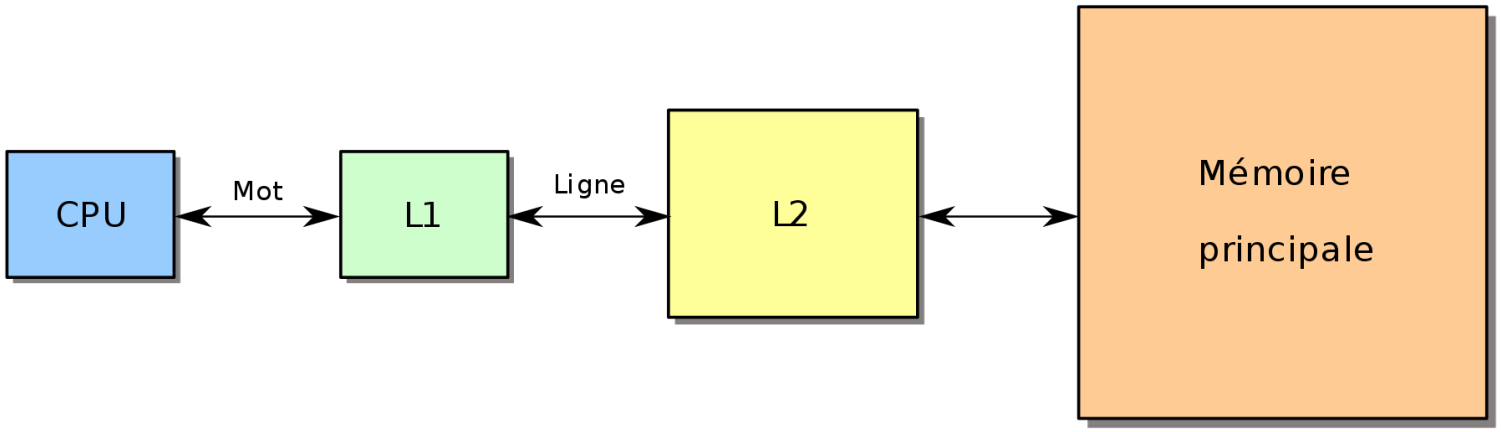
Fragmentation

- La fragmentation indique la mauvaise utilisation de la mémoire (gâchis)
  - Fragmentation externe



    - Pas possible de *défragmenter* une fois les adresses retournées par `malloc()` à l'application !
  - Fragmentation interne : espace utilisé pour stocker les métadonnées et perdu suite au padding utilisé pour respected la contrainte d'alignement
- On souhaite limiter les deux types le plus possible

Principe de cache



- Un processeur n'accède pas directement à la mémoire principale
  - Trop lente ! Accès en lecture : centaines de cycles processeur
  - Le processeur passerait son temps à attendre la mémoire principale …
- Cache : mémoire rapide (et chère) proche du processeur
  - Les données récemment utilisées sont stockées dans la mémoire cache
  - Si une donnée n'est pas présente dans le cache, elle y est rapatriée depuis la mémoire principale
  - Granularité du cache : *ligne de cache*, par exemple 64 octets (512 bits)

Localité spatiale

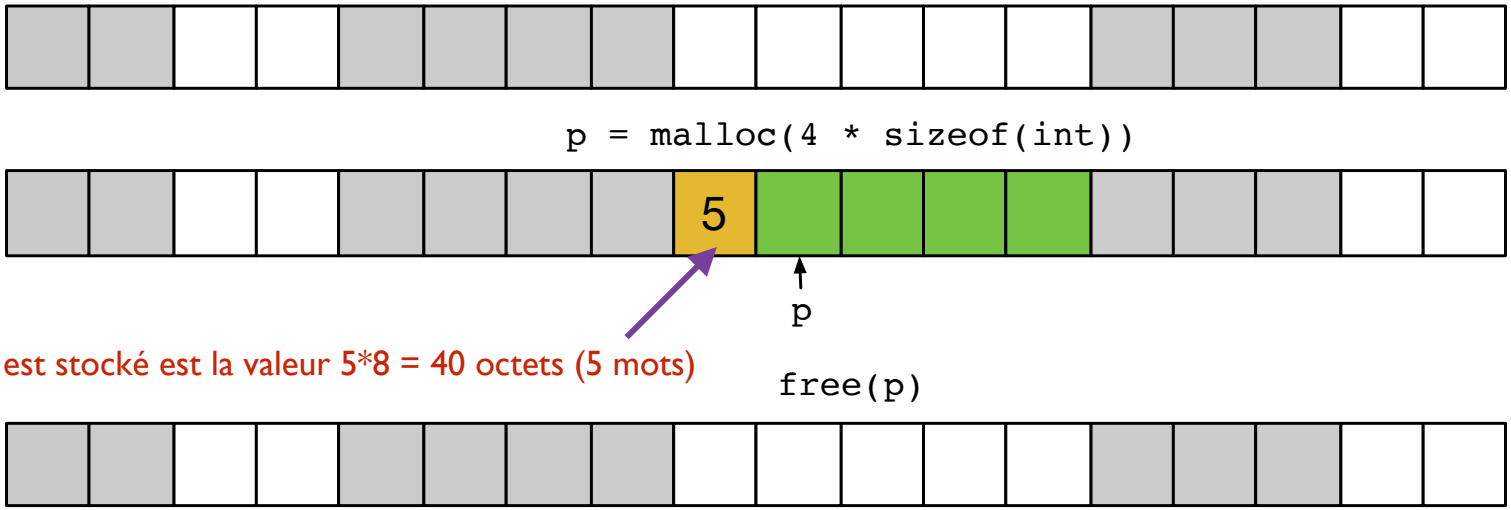
- Le principe de cache fonctionne grace à la localité
  - **Localité spatiale** : l'accès à une donnée est généralement suivie d'accès à des données contiguës
    - Exemple : parcours itératif d'un tableau
  - Localité temporelle : une donnée lue/écrite récemment est susceptible d'être lue/écrite dans un futur proche
    - Exemple : un itérateur de boucle
- La mémoire allouée dynamiquement doit idéalement favoriser les propriétés de localité (surtout spatiale)
  - Blocs alloués successivemnt dans le temps proches en mémoire
  - Maximiser les chances d'être sur une ligne de cache déjà lue
  - Exemple : les éléments d'une liste chaînée

Algorithmes de gestion de mémoire dynamique

- Nous allons seulement étudier des exemples simples : les algorithmes utilisés dans Linux sont bien plus complexes
  - Mais ils vont nous permettre de comprendre le compromis entre les différents critères :
    - Efficacité d'exécution
    - Efficacité d'utilisation de la mémoire
    - Localité spatiale
    - Bonus : robustesse (`free(ptr)` avec `ptr` non retourné par malloc ou déjà libéré : détecté ?)
- Liste implicite ou explicite

Stockage des métadonnées

- Simplification : mémoire divisée en **mots** contenant un entier ou un pointeur (64 bits)
- Métadonnées pour un **bloc**
  - Taille de ce bloc
  - Drapeau : bloc alloué ou bloc libre ?
- On utilise des mots pour stocker les métadonnées d'un bloc

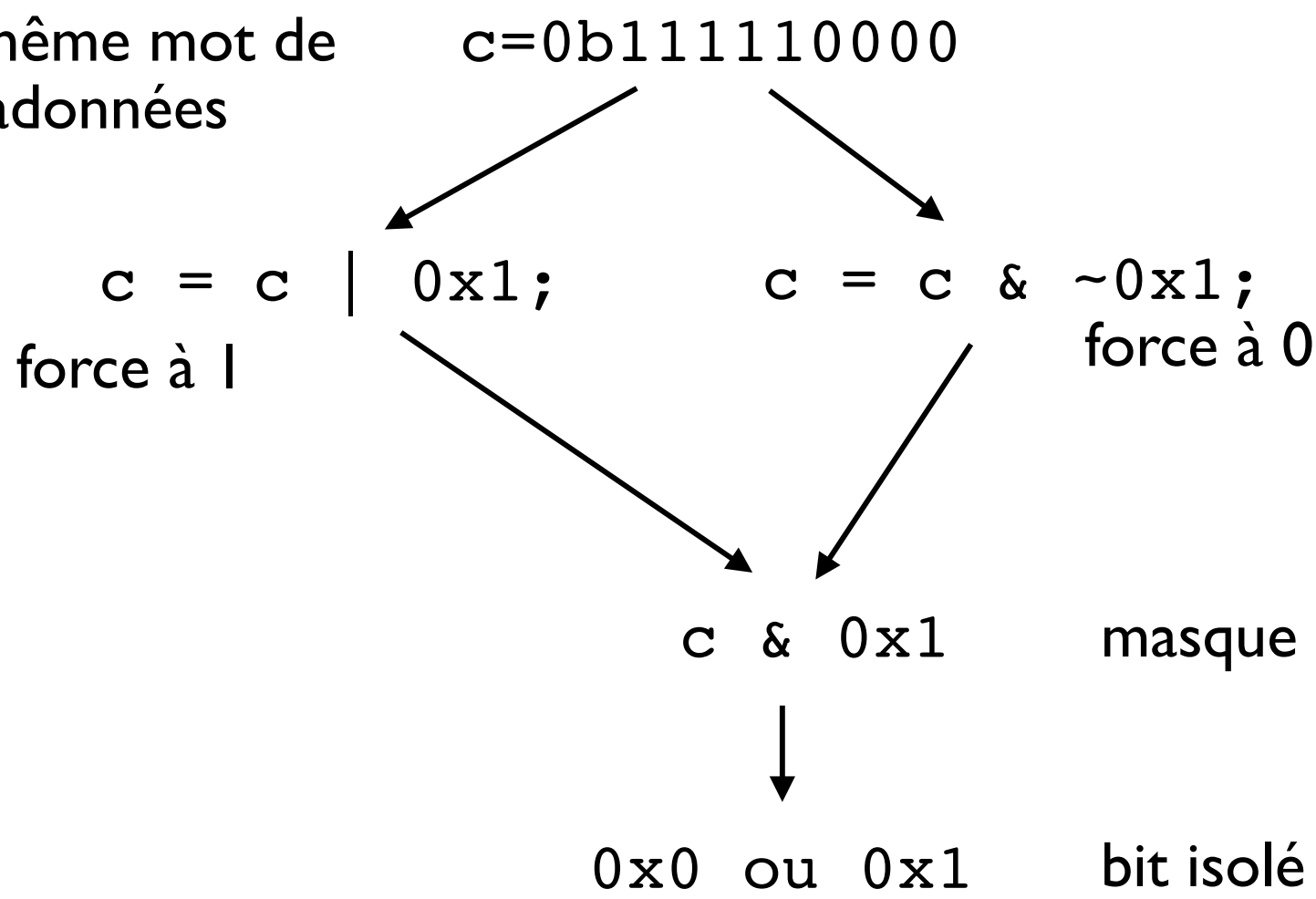


Opérations binaires en pratique

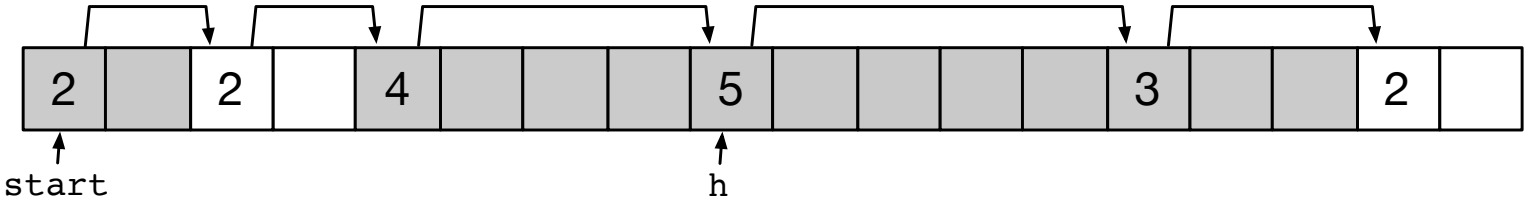
- On alloue des blocs (zones contiguës de mots) dont la taille est un multiple de 2^x $(x>0)$
  - Incluant le (ou les) mots de métadonnées
  - Le bit de poids faible d'une adresse ou d'un décalage entre deux blocs est donc toujours à 0
  - ex. `0x5FDC810` ; `0x458` ;
    `0x1F0` = `0b111110000`
- Le compteur, nombre d'octets d'un bloc termine donc toujours par (au moins) un bit à 0
- On peut utiliser ce bit de poids faible pour stocker un bit *drapeau* qui indique si le bloc est libre ou alloué

Opérations binaires en pratique

compteur/taille combinés dans le même mot de métadonnées



Liste implicite

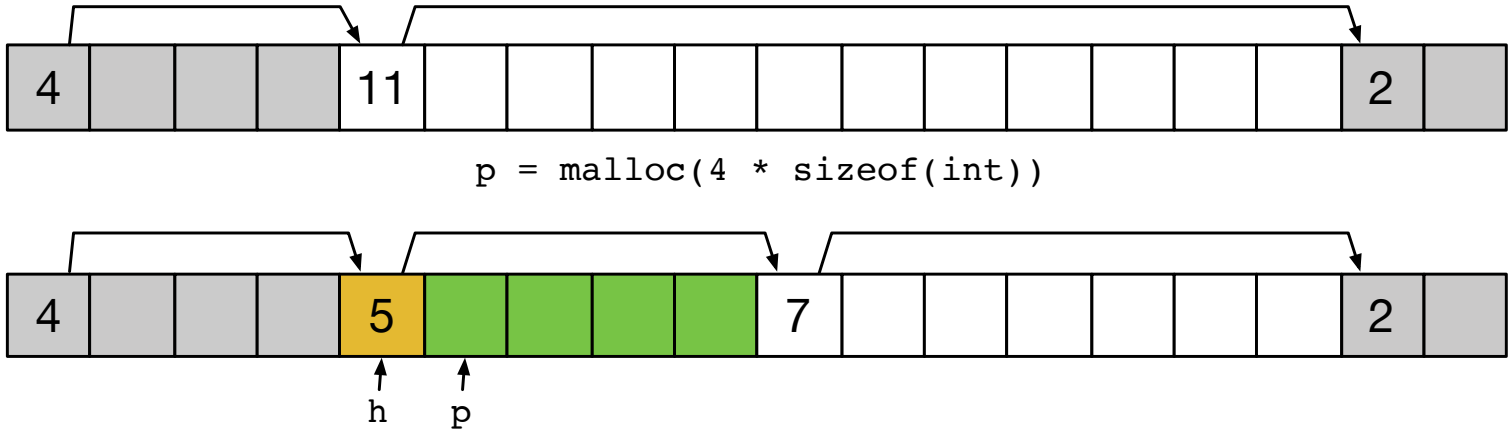


- Chaque bloc commence par un mot de métadonnées (header)
  - Taille du bloc (en octets, #ints pour la figure)
  - Type de bloc {libre,alloué} dans le bit de poids faible
- Parcours de l'ensemble des blocs pour trouver un bloc libre

```
h = start;
while (h < end &&              // fin de liste ?
       ((*h & 0x1) != 0 ||     // déjà alloué
        *h <= len))            // trou trop petit
    h = h + (*h & ~0x1);       // progresse vers le prochain bloc
```

Scission d'un bloc

- Le bloc trouvé peut être plus grand que la taille de bloc demandé : il faut le scinder



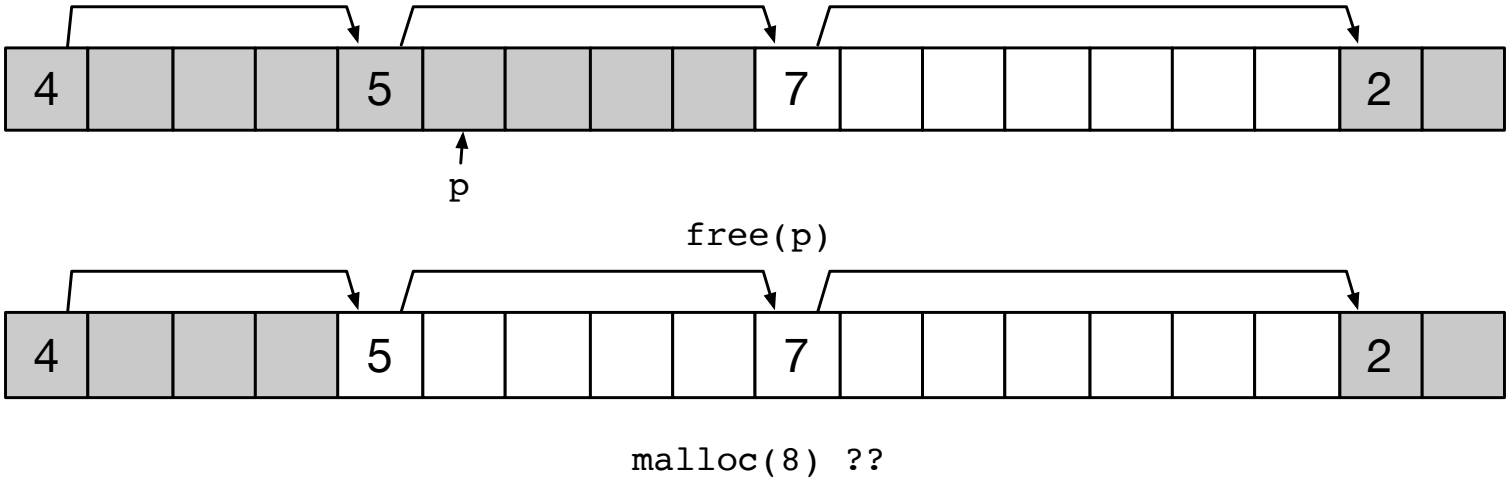

```
void place_block(ptr h, int len) {
  int newsize = len + 1;                  // ajoute 1 mot pour le header
  int oldsize = *h & ~0x1;                // récupère taille actuelle sans bit de poids faible
  *h = newsize | 0x1;                     // nouvelle taille avec bit de poids faible à 1
  if (newsize < oldsize)                  // s'il reste de la place ...
    *(h + newsize) = oldsize - newsize; // nouveau bloc vide avec la taille restante et bit PF à 0
}
```

Libération d'un bloc avec `free(p)`

- La méthode la plus simple est de passer le bit de poids faible du mot à l'adresse `(p-1)` à 0
  - Si l'adresse p n'est pas un début de bloc valide : risque de corruption de données (bug compliqué à détecter, reproduire et résoudre !)
  - On peut parcourir la liste et vérifier si (p-1) est un bloc de métadonnées valide pour un bloc alloué : coût O(N) vs. coût O(1)
    - En général, on le fait pas car c'est considéré comme trop coûteux
    - Illustration d'un compromis entre performance et robustesse aux erreurs de programmation !

Libération d'un bloc avec `free(p)`

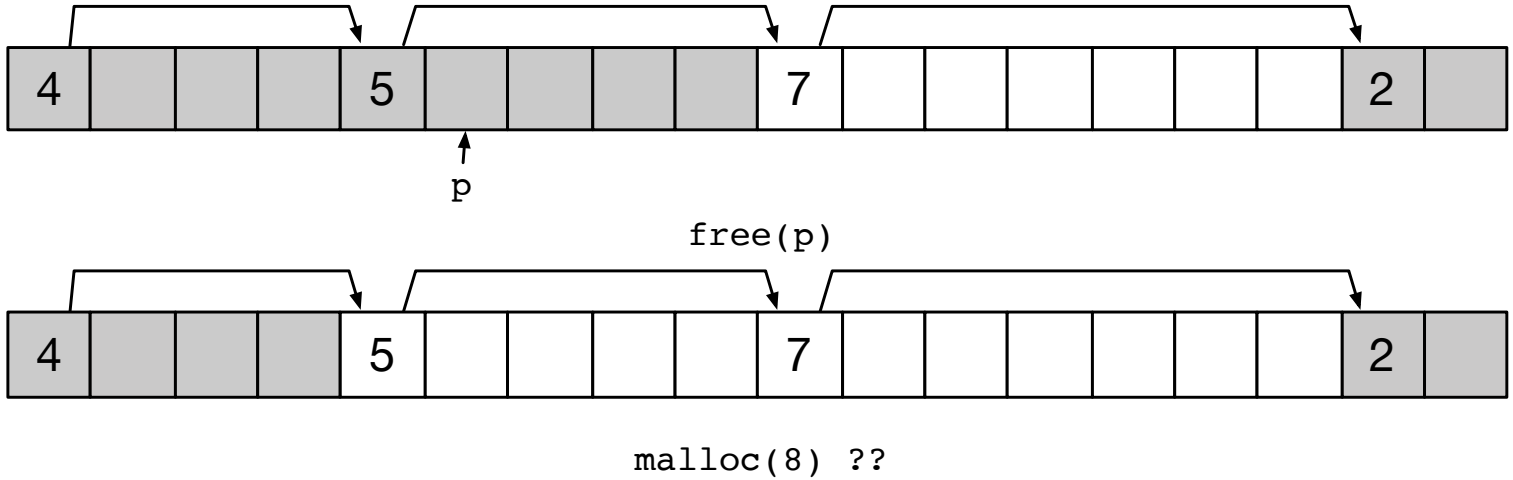
- La méthode la plus simple est de passer le bit de poids faible du mot à l'adresse `(p-1)` à 0
  - ☹ Problème de *fausse fragmentation*
  - ➡ Plusieurs blocs libres qui se suivent



Solutions à la fausse fragmentation

1. Laisser les blocs libres contigües tels quels
   - Approche paresseuse (procrastinatrice)
   - Les blocs peuvent être fusionnés
     - Lors d'un prochain `malloc()` qui cherche un bloc libre
     - Si la mémoire disponible est < à un seuil
2. ou fusionner les blocs immédiatement

Fusion de blocs lors du `free(p)`

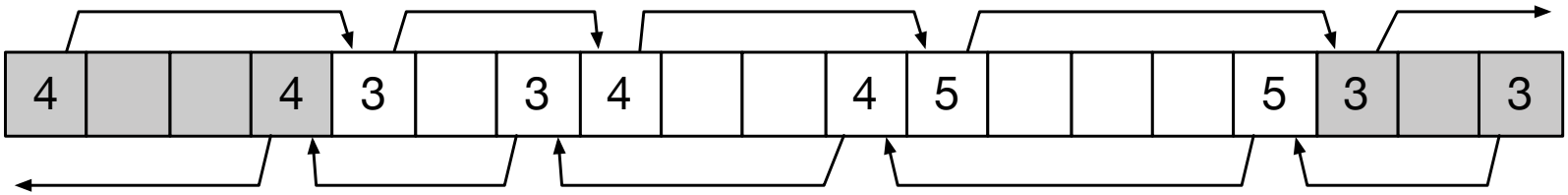



```
void free_block(ptr h) {
  *h = *h & ~0x1;            // remise à 0 du drapeau
  next = h + *h;             // calcul de l'adresse du header du bloc suivant
  if ((*next & 0x1) == 0)    // si ce bloc est aussi libre ...
    *h = *h + *next;         // combiner les blocs
}
```

Problème : on peut fusionner un bloc libre avec son successeur, mais pas avec son prédécesseur ! (Liste chaînée simple)

Fusion bi-directionnelle

- Solution 1 : `free(p)` conserve l'historique des blocs libres (contigües) vus précédemment lors du parcours de la liste
  - Ne consomme pas de mémoire supplémentaire du heap
  - Mais algorithme de parcours un peu plus complexe
- Solution 2 : liste doublement chaînée
  - Le header est présent à la fois au début et à la fin du bloc (utilisation de 2 mots à la place d'un seul)
  - Parcours dans les deux sens possible

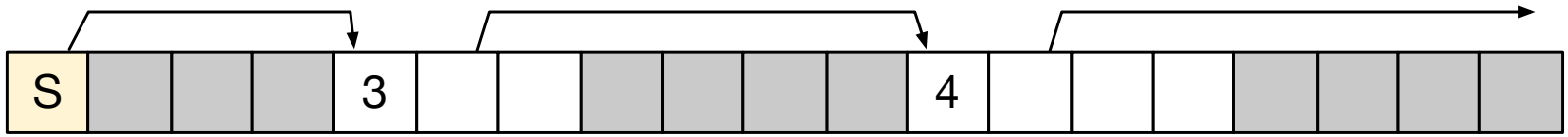


Politiques de placement

- Est-il sage de choisir le premier bloc libre de taille suffisante rencontré lors du parcours ? (Politique "first fit")
  - 😀 Rapide
  - ☹️ Entraîne de la fragmentation
  - ☹️ Localité faible, empirant au fur et à mesure de l'exécution du programme
- Politique "next fit" : comme "first fit" mais démarre la parcours depuis le dernier bloc alloué
  - 🙂 Meilleures propriétés de localité
  - 😱 Fragmentation très élevée
- Politique "best fit" : parcours intégral de la liste pour trouver le bloc libre le plus petit possible pouvant accueillir le nouveau bloc
  - 😎 Optimal en termes de fragmentation
  - 😟 Localité médiocre
  - 😱 Coût d'exécution élevé

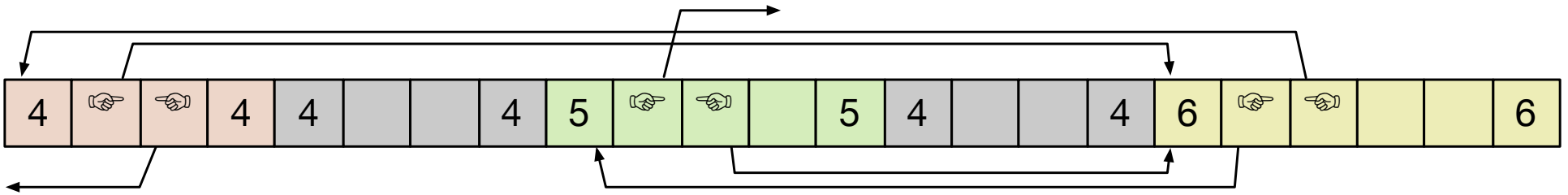
Liste explicite

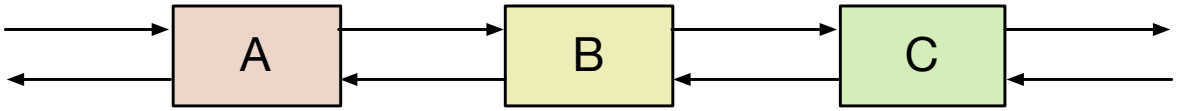
- Alternative : lier les blocs vides entre eux, et plus l'ensemble des blocs
  - Premier mot d'un bloc vide ou alloué = sa taille
  - Deuxième mot d'un bloc vide (seulement) = pointeur vers le prochain bloc libre
  - Pointeur initial dans le premier bloc du segment
  - Parcours de la liste pour trouver bloc first/next/best-fit



Liste explicite doublement chaînée

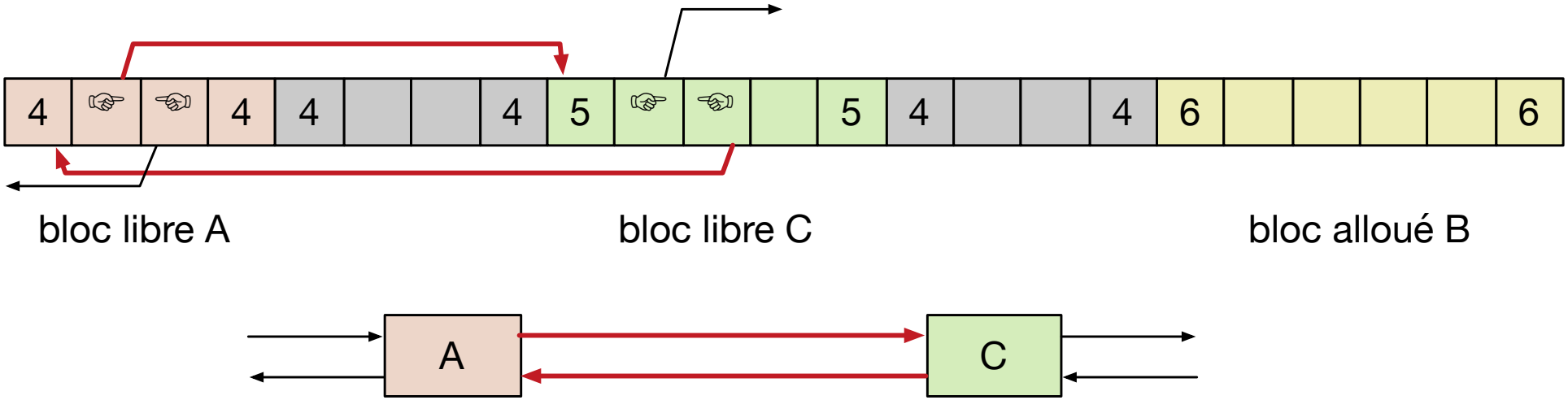
- En pratique, utilisation d'une liste doublement chaînée
  - Mot 1 = compteur (y compris métadonnées)
  - Mot 2 = pointeur vers bloc suivant
  - Mot 3 = pointeur vers bloc précédent
- L'ordre dans la liste chaînée ne respecte pas nécessairement l'ordre des adresses des blocs en mémoire !
  - Nécessaire de dupliquer le header en début et fin du segment pour permettre la fusion bidirectionnelle
  - Taille minimum d'un bloc = 4 mots de métadonnées





Allocation d'un bloc

- Nécessite de maintenir la liste doublement chaînée
  - Modification du pointeur successeur du bloc A
  - et du pointeur prédécesseur du bloc C



Libération d'un bloc

- Plus complexe qu'avec une liste implicite
  - L'ordre des blocs dans la liste ne suit pas forcément l'ordre de leurs adresses en mémoire
- Solution 1 (*address-ordered*)
  - Forcer la correspondance des deux ordres
  - ☹ Nécessite de parcourir la liste pour insérer le bloc libéré au bon endroit dans la liste : coût O(N)
  - 😀 Opération de fusion simplifiée !
- Solution 2 (*LIFO — Last-In-First-Out*)
  - Insérer le bloc libéré au début de la liste des blocs vides
  - 😀 Coût de la libération en O(1)
  - ☹ Fusion plus complexe

Utilisation de listes multiples

- Une seule liste :
  - Temps de recherche d'un bloc augmente linéairement avec le nombre de blocs libres
  - Temps de recherche de grand bloc + long
- Plusieurs listes
  - Selon la taille des blocs libres
  - 1, 2, 3, 4 puis puissances de 2 : 8, 16, 32, 64 …
  - Recherche d'un bloc de taille 18 dans la liste 16
    - Tous les blocs libres de taille 9 à 16
  - Politique address-ordered alors très coûteuse !

Conclusion

Conclusion

- Gestion de la mémoire en C
  - Différents *segments* : text, data, stack, heap
- Allocation et libération de mémoire dans le heap
  - Compromis entre complexité, performance, utilisation de la mémoire
  - Les algorithmes utilisés dans les systèmes UNIX modernes et Linux prennent de nombreux autres critères en compte ! (sujet de recherche encore actuel)
  - Illustration de la différence entre mécanisme et politique
- À vous de jouer : projet P0, implémentation de `malloc()` et `free()`
  - Travail préparatoire en séance (TD)
  - Mise en œuvre et tests via Inginious, rapport (1 page) à rendre avec illustrations et explications
  - Travail en binôme (copie entre binômes = triche)