Système

La mémoire

Gilles Roussel

Gilles.Roussel@univ-mlv.fr

http://igm.univ-mlv.fr/ens/Licence/L2/2011-2012/Systeme/

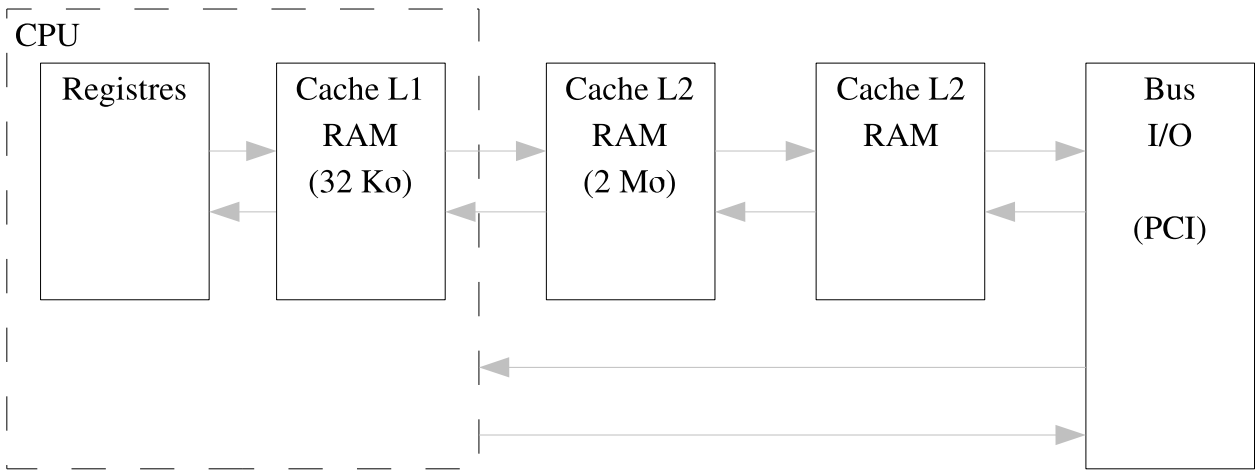
Licence 2

9 avril 2012

Qu'est-ce que l'on appelle la mémoire (vive) ?

- *Random Access Memory* (RAM)
- Ensemble de transistors utilisés comme des condensateurs : capacité résiduelle utilisée pour stocker l'information
- Besoin de rafraîchir la mémoire régulièrement (latence)
- Débit moyen (en 2008) 1000 Mo/s contre :
  - 200 Mo/s pour une interface de disque dur (sans compter la latence)
  - 5000 Mo/s pour le cache de niveau 2
  - 10000 Mo/s pour le cache de niveau 1 et les registres

La hiérarchie mémoire



La hiérarchie mémoire

matrix.c et matrix2.c

```
/* matrix.c */
#include <stdlib.h>
#define SIZE 1024
int main(int argc, char * argv[]) {
  int i,j, matrix[SIZE][SIZE];
  for (i=0;i<SIZE;i++) {
    for (j=0;j<SIZE;j++) {
      matrix[i][j] = i*j;
    }
  }
  exit(EXIT_SUCCESS);
}
/* matrix2.c */
#include <stdlib.h>
#define SIZE 1024
int main(int argc, char * argv[]) {
  int i,j, matrix[SIZE][SIZE];
  for (i=0;i<SIZE;i++) {
    for (j=0;j<SIZE;j++) {
      matrix[j][i] = i*j;
    }
  }
  exit(EXIT_SUCCESS);
}
```

La hiérarchie mémoire

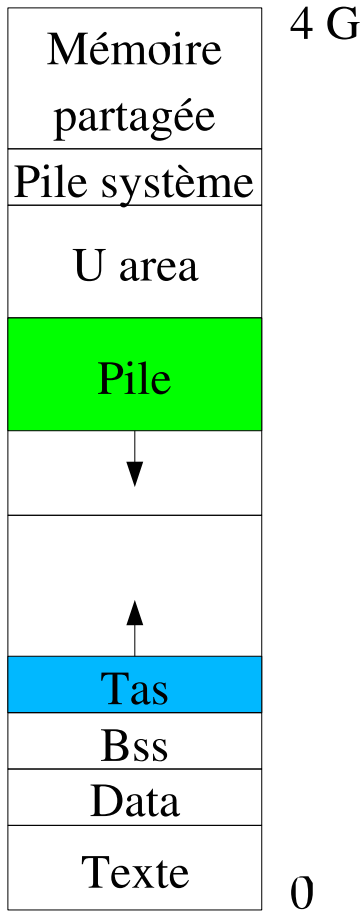

```
# time ./matrix
real    0m0.062s
user    0m0.022s
sys     0m0.009s
# time ./matrix2
real    0m0.181s
user    0m0.134s
sys     0m0.010s

#  valgrind --tool=cachegrind ./matrix
==9841== Cachegrind, an I1/D1/L2 cache profiler for x86-linux.
...
==9841== D   refs:       7,397,217  (6,332,576 rd + 1,064,641 wr)
==9841== D1  misses:        66,650  (      884 rd +    65,766 wr)
==9841== L2d misses:        65,781  (      150 rd +    65,631 wr)
==9841== D1  miss rate:        0.9% (      0.0%   +       6.1%  )
==9841== L2d miss rate:        0.8% (      0.0%   +       6.1%  )
...

#  valgrind --tool=cachegrind ./matrix2
==9841== Cachegrind, an I1/D1/L2 cache profiler for x86-linux.
...
==9870== D   refs:       7,397,215  (6,332,574 rd + 1,064,641 wr)
==9870== D1  misses:     1,049,690  (      884 rd + 1,048,806 wr)
==9870== L2d misses:        65,781  (      150 rd +    65,631 wr)
==9870== D1  miss rate:      14.1% (      0.0%   +      98.5%  )
==9870== L2d miss rate:        0.8% (      0.0%   +       6.1%  )
...
```

Accès à la mémoire

- Processus n'accède pas directement à la mémoire physique
- Notion de mémoire logique (virtuelle) : tout se passe comme si le processus avait « toute » la mémoire pour lui tout seul
- Taille de la mémoire virtuelle ne dépend pas de la taille de la mémoire physique, mais de la taille des pointeurs (32bits=4Go)
- Association entre adresse mémoire logique et adresse mémoire « physique » réalisée par matériel *Memory Management Unit* (MMU) configuré par le système
- Association par page (4k sur les PC `getpagesize()`)
- Permet au système de modifier le contenu de la mémoire physique de façon transparente pour les processus

Organisation de la mémoire

- Espace adressable 4G pour processeur 32bits
- Texte : code du programme
- Data : variables globales initialisées
- Bss : variables globales non initialisées
- Tas : réservation mémoire avec `malloc`
- Pile : variables locales (automatique)
- U-area : informations processus (fichiers ouverts, répertoire de travail, etc.)
- Pile système : utilisée par appels systèmes
- Mémoire partagée



Organisation de la mémoire

memory.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
char data = 'c';
char *bss;
int main(int argc, char * argv[]) {
  int pile;
  bss = (char*) malloc(sizeof(char));
  printf("texte\t%p\n",main);
  printf("data\t%p\n",&data);
  printf("bss\t%p\n",&bss);
  printf("tas\t%p\n",bss);
  printf("pile\t%p\n",&pile);
  for(;;);
  exit(EXIT_SUCCESS);
}
```

Organisation de la mémoire


```
# memory
texte   0x80481e8
data    0x80a506c
bss     0x80a6e08
tas     0x849a2e8
pile    0xbffa4f94
# ps -C memory
  PID TTY          TIME CMD
20756 pts/1    00:00:10 memory
# cat /proc/20756/maps
08048000-080a5000 r-xp 00000000 03:09 34206 memory texte
080a5000-080a6000 rw-p 0005d000 03:09 34206 memory data
080a6000-080a7000 rw-p 080a6000 00:00 0             bss
0849a000-084bb000 rw-p 0849a000 00:00 0             tas
b7f0d000-b7f0e000 rw-p b7f0d000 00:00 0             ??
bffa4000-c0000000 rw-p bffa4000 00:00 0             pile
ffffe000-fffff000 ---p 00000000 00:00 0             ??
```

Organisation de la mémoire

Mécanisme de mémoire virtuelle permet que :

- certaines pages de la mémoire soient en lecture seule, exécution, etc. : mécanisme au niveau matériel
  - violation récupérée par le système et signalée au programme : erreur de segmentation (`SIGSEGV`)
- certaines pages soient inexistantes
- le texte soit partagé entre plusieurs processus
- certaines pages de la mémoire soient placées temporairement sur le disque : partition d'échange (*swap*) ou code

Utilisation de la mémoire

- `top` donne une idée de l'utilisation de la mémoire

```
top - 21:17:22 up 27 min,  2 users,  load average: 0.00, 0.04, 0.12
Tasks:  98 total,   1 running,  97 sleeping,   0 stopped,   0 zombie
Cpu(s):  7.0% us,  1.7% sy,  0.0% ni, 91.1% id,  0.0% wa,  0.3% hi,  0.0% si
Mem:   636116k total,   535476k used,   100640k free,    23288k buffers
Swap:  1028120k total,        0k used,  1028120k free,   310828k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 6549 roussel   15   0  117m  56m  33m S  0.7  9.1   0:13.29 acroread
 5753 root      15   0  178m  30m 7376 S  2.7  4.9   0:27.28 X
 6533 roussel   16   0  103m  28m  16m S  0.0  4.5   0:06.20 firefox-bin
...
```

Alignement et promotion entière

- Les entiers du C sont codés sur un mot machine
- Le processeur manipule uniquement des mots
- Certains processeurs peuvent accéder uniquement à un mot en mémoire s'il est aligné, c'est-à-dire si son adresse est égale à un multiple de la taille d'un mot (`Bus error`)
- Les `char` et les `short` sont « promus » en entiers avant d'être manipulés par le processeur, leur adresse en mémoire n'est pas importante
- Les structures alignent par défaut les entiers (sans option `-fpack-struct` de gcc)

Alignement et promotion entière

alignment.c

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/times.h>
#define PAGE_SIZE 1024

typedef struct {
  char a;
  int b;
 char c;
} aligned;

typedef struct {
  char a;
  char c;
  int b;
} __attribute__((packed)) packed;

int main(int argc, char * argv[]) {
  printf("aligned %ld\n",sizeof(aligned));
  printf("packed %ld\n",sizeof(packed));
  exit(EXIT_SUCCESS);
}
```

```
# ./alignment
aligned 8
```

Texte, data, bss

- Texte : fichier directement accessible en mémoire ; chargé à la demande
- Zones mémoires déterminées au moment la compilation

```
# readelf -e memory
En-têtes de section:
[Nr] Nom   Type     Adr      Décala. Taille ES Fan LN Inf Al
[ 2] .text PROGBITS 08048100 000100  0476e0 00  AX  0   0 32
...
[15] .data PROGBITS 080a5060 05d060  000cd4 00  WA  0   0 32
[16] .bss  NOBITS   080a5d40 05dd34  001284 00  WA  0   0 32
...
```

Pile

- Allouée au cours de l'exécution du programme à chaque appel de fonction
- Désallouée lors du retour de la fonction

pile.c

```
#include <stdlib.h>
#include <stdio.h>
void f(int i) {
  printf("%p ",&i);
  if (i<3) { f(i+1); }
}
int main(int argc, char * argv[]) {
  f(0);
  exit(EXIT_SUCCESS);
}
```

```
# pile
0xbfe0e904 0xbfe0e8e4 0xbfe0e8c4 0xbfe0e8a4
```

Tas

- Géré par l'utilisateur
- Deux interfaces :
  - bas niveau appel système et `libc` : `brk()` et `sbrk()`
  - haut niveau définie dans la `libc` : `malloc()`, `calloc()`, `realloc()` et `free()`

Interface de bas niveau

- `brk()` prend en argument l'adresse du haut du tas et retourne `0` ou `-1` en cas d'erreur (pas utilisable sans connaissance supplémentaire)
- `sbrk()` prend en argument un nombre d'octets à ajouter au tas. Elle retourne l'adresse de début de nouvelle zone réservée ou `(void*)-1` en cas d'erreur
- `sbrk(0)` retourne l'adresse du haut du tas courant

Un malloc() simpliste

allocateur.c

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

void *malloc(size_t size) {
  // Pour l'alignement
  int alloc_size = ((size-1)/sizeof(int)+1+1)*sizeof(int);
  int *ptr = sbrk(alloc_size);
  printf("malloc\n");
  if ((int*)-1 == ptr) {
    return NULL;
  }
  // Pour se souvenir de la taille allouée
  *ptr=alloc_size;
  return ptr+1;
}
```

Parenthèse : « memset(), memcpy() et memmove() »

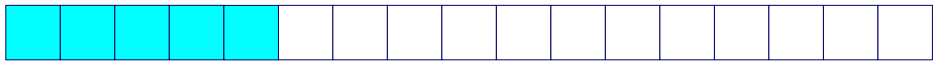
- Déclarées dans `string.h`
- `void *memset(void *s, int c, size_t n)`
  copie la valeur de l'octet de poids faible de c dans les `n` premiers octets du tableau d'octets pointé par `s` et retourne `s`
- `void *memcpy(void *dest, const void *src, size_t n)`
  copie les `n` premiers octets de `src` dans `dest` et retourne `dest`. Ne marche pas si `src` et `dest` se recouvrent
- `void *memmove(void *dest, const void *src, size_t n)`
  même sémantique que `memcpy()`, moins efficace mais marche si `src` et `dest` se recouvrent

Interface haut niveau

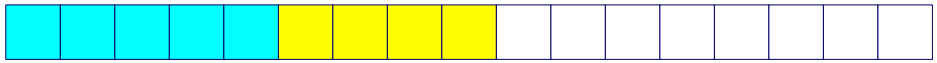
- `void *malloc(size_t size))` retourne pointeur sur multiple de 8 (libc) ou `NULL` en cas d'erreur
- `void *calloc(size_t nmemb, size_t size)` même chose que `malloc()` sauf que le contenu est mis à `0` ; équivalent(?) à `malloc()` suivi de `memset()`
- `void free(void *ptr)` libère mémoire allouée avec `malloc()`, `calloc()` ou `realloc()`
- `void *realloc(void *ptr, size_t size)`
  - si ancien pointeur `NULL` même chose que `malloc()`
  - si nouvelle taille est `0` même chose que `free()`
  - si nouvelle taille inférieure à l'ancienne : libération mémoire ($\neq$ `free()`)
  - si erreur `ptr` encore valide
  - sinon alloue `size` octets, copie le contenu de `ptr` ($<$ `size`) et désalloue `ptr` (`free()`)

Exemple d'allocation


```
p1 = malloc(5*sizeof(int))
```

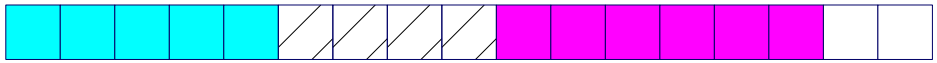


```
p2 = malloc(4*sizeof(int))
```

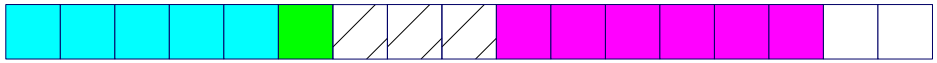


```
p3 = malloc(6*sizeof(int))
```

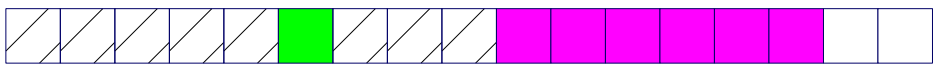
```
free(p2)
```



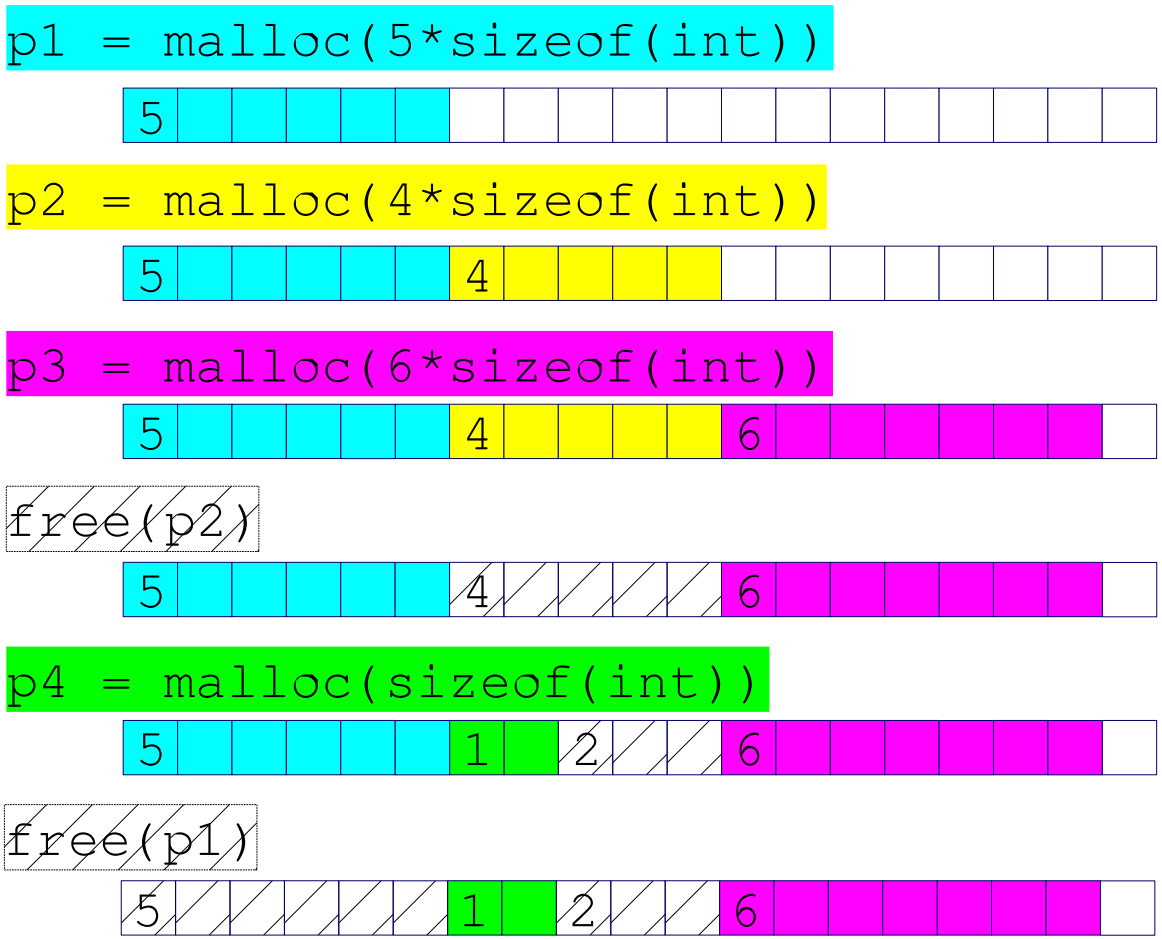
```
p4 = malloc(sizeof(int))
```



```
free(p1)
```

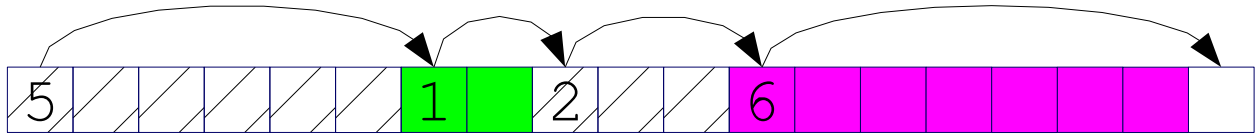


Comment connaître la taille des allocations ?

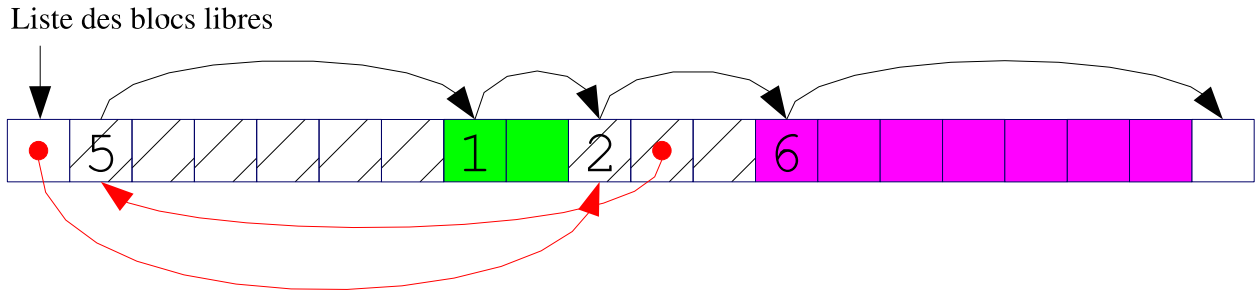


Comment trouver un bloc libre ?

- Liste implicite : utiliser le bit de poids fort de la taille du bloc pour marquer les blocs libres et parcourir tous les blocs pour trouver un bloc libre de la bonne taille, recherche en temps linéaire ($O(N)$) en la *taille du tas*



- Liste explicite : utiliser les blocs libres pour construire une liste des blocs libres, recherche en temps linéaire ($O(n)$) en la *taille de la liste des blocs libres*



Comment gérer la liste des blocs libres ?

- Insertion des blocs libérés en tête (LIFO):
  - insertion en temps constant $O(1)$
  - recherche d'un bloc libre d'une certaine taille en temps linéaire $O(n)$ en la taille de la liste
  - augmente la fragmentation
- Insertion des blocs libérés dans l'ordre des adresses :
  - insertion et recherche en temps linéaire $O(n)$ en la taille de la liste
  - facilite la défragmentation
- Une listes de blocs libres par taille :
  - mêmes complexités dans le pire des cas que LIFO
  - meilleur en moyenne
- De nombreuses autres stratégies existent

Chercher les 5 erreurs

erreurs.c

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <string.h>
 4 #define TEST_STR "test"
 5 int main(){
 6   char *ptr1, *ptr2;
 7   ptr1 = malloc(strlen(TEST_STR));
 8   ptr2 = malloc(strlen(TEST_STR));
 9   if (NULL == ptr1 || NULL == ptr2){
10     perror("malloc");
11     exit(EXIT_FAILURE);
12   }
13   strcpy(ptr1,TEST_STR);
14   printf("%s\n",ptr2);
15   strcpy(ptr2,ptr1);
16   printf("%s\n",ptr2);
17   free(ptr2);
18   exit(EXIT_SUCCESS);
19 }
```

Trouver les 5 erreurs


```
# valgrind --tool=memcheck ./erreurs
==11490== Memcheck, a memory error detector for x86-linux.
...
==11490== Invalid write of size 1
==11490==    at 0x1B905440: strcpy (mac_replace_strmem.c:199)
==11490==    by 0x80484E1: main (erreurs.c:15)
...
==11490== Conditional jump or move depends on uninitialised value(s)
==11490==    at 0xC35D09: strlen (in /lib/tls/libc-2.3.5.so)
==11490==    by 0x80484F4: main (erreurs.c:16)
...
==11490== Invalid read of size 1
==11490==    at 0x1B905438: strcpy (mac_replace_strmem.c:198)
==11490==    by 0x8048505: main (erreurs.c:17)
...
==11490== Invalid write of size 1
==11490==    at 0x1B905440: strcpy (mac_replace_strmem.c:199)
==11490==    by 0x8048505: main (erreurs.c:17)
...
==11490== Invalid read of size 4
==11490==    at 0xC35CB9: strlen (in /lib/tls/libc-2.3.5.so)
==11490==    by 0x8048518: main (erreurs.c:18)
...
==11490== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 12 from 1)
==11490== malloc/free: in use at exit: 4 bytes in 1 blocks.
==11490== malloc/free: 2 allocs, 1 frees, 8 bytes allocated.
...
```