

Utilisation des systèmes UNIX

Benjamin Negrevergne

PSL University – Paris Dauphine – Équipes *MILES*



Gestion mémoire

Au niveau matériel, il n'y a pas

- de variables, de nom de variable ou de types
(i.e. pas d'entiers, pas de flottants, pas de chaîne de caractères ou de tableaux)
- de “zone vide” ou “zone écrite”
- de frontières qu'il serait possible positionner arbitrairement

Gestion mémoire

Au niveau matériel, il n'y a pas

- de variables, de nom de variable ou de types
(i.e. pas d'entiers, pas de flottants, pas de chaîne de caractères ou de tableaux)
- de "zone vide" ou "zone écrite"
- de frontières qu'il serait possible positionner arbitrairement

Il y a seulement

- des bits (0/1) regroupés en octets (bytes)
 - chaque octet a un adresse mémoire
 - on peut modifier la valeur d'un ou plusieurs octets avec LOAD/STORE
- de grosses contraintes de performance
dues aux problèmes d'alignement mémoire et aux mémoires caches

Gestion mémoire (2)

Pour assurer l'exécution d'un programme, il faut:

- réserver des octets pour stocker la valeur de chaque variable
- éviter le chevauchement des valeurs dans la mémoire
- éviter de trop fragmenter la mémoire
- libérer la mémoire inutile

Gestion mémoire (2)

Pour assurer l'exécution d'un programme, il faut:

- réserver des octets pour stocker la valeur de chaque variable
- éviter le chevauchement des valeurs dans la mémoire
- éviter de trop fragmenter la mémoire
- libérer la mémoire inutile

Idée: deux régions qui vont être gérées différemment:

- Pile: région ordonnée, allocation/libération gérée compilateur
- Tas: région fragmentée, allocation/libération gérée par le programmeur (aidé de malloc)

Pile/Tas

Deux régions de l'espace des adresses utilisées pour stocker les données nécessaires à l'exécution du programme.

Extrait de `/proc/<pid>/maps`

1	5603b8e22000-5603b8e43000	rw-p	00000000	00:00	0	[heap]
2	7ffda2e9e000-7ffda2ebf000	rw-p	00000000	00:00	0	[stack]

Taille initiale:

- 1.3 Go pour le tas.
- 120 octets pour la pile.

Pile/Tas

Deux régions de l'espace des adresses utilisées pour stocker les données nécessaires à l'exécution du programme.

Extrait de `/proc/<pid>/maps`

```
1 5603b8e22000-5603b8e43000 rw-p 00000000 00:00 0 [heap]
2 7ffda2e9e000-7ffda2ebf000 rw-p 00000000 00:00 0 [stack]
```

Taille initiale:

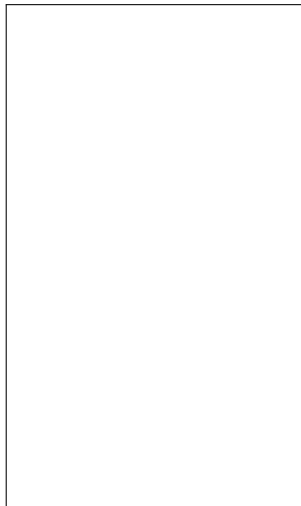
- 1.3 Go pour le tas.
- 120 octets pour la pile.

```
1 void f(int a){
2   int b = 2;
3   int *c = malloc(sizeof(int));
4   printf("%p, %p, %p, %p\n",
5         &a, &b, &c, c);
6 }
```

```
1 &a: 0x7ffda2ebcc0c -> pile
2 &b: 0x7ffda2ebcc1c -> pile
3 &c: 0x7ffda2ebcc10 -> pile
4 c: 0x5603b8e222a0 -> tas
```

Pile/Tas

```
1 &a: 0x7ffda2ebcc0c -> pile @12  
2 &b: 0x7ffda2ebcc1c -> pile @28  
3 &c: 0x7ffda2ebcc10 -> pile @16  
4 c: 0x5603b8e222a0 -> tas
```



(note: schéma à compléter)

Pile (stack)

- Utilisée pour les variables locales et les arguments de fonctions.
- Allocation et libération implicite (gérée par le **compilateur**)
 - La mémoire est allouée au début de l'exécution de la fonction
 - La mémoire est libérée à la fin de l'exécution de la fonction
- Taille de la mémoire à allouer est déterminée à partir du types des variables
- ⚠ Impossible de prolonger la durée de vie d'une variable au delà du contexte d'exécution de la fonction
- Pas de fuite mémoire possible
- Pas de fragmentation mémoire

Tas (heap)

- Allocation/libération explicite (gérée **par le programmeur** via `malloc` et `free`)
- Taille de la mémoire à allouer est déterminée par le programmeur
- Possibilité de prolonger la durée de vie d'une allocation arbitrairement
- La fragmentation est inévitable
- ⚠ possibilité de fuite mémoire

Tas (heap)

- Allocation/libération explicite (gérée **par le programmeur** via `malloc` et `free`)
 - Taille de la mémoire à allouer est déterminée par le programmeur
 - Possibilité de prolonger la durée de vie d'une allocation arbitrairement
 - La fragmentation est inévitable
 - ⚠ possibilité de fuite mémoire
- ▶ Préférer les allocations sur la pile lorsque c'est possible!

```
1 readbuf(...){
2     char *buf = malloc(sizeof(char)*BUF_SIZE); //inutile
3     read(fd, buf, BUF_SIZE);
4     printf("buf : %s\n", buf);
5     free(buf);
6 }
```

Pile et contexte d'exécution d'une fonction

```
1 int power(int x, int n){  
2     int res;  
3     if ( n == 0)  
4         res = 1;  
5     else  
6         res = x * power(x, n - 1);  
7     return res;  
8 }
```

Pile et contexte d'exécution d'une fonction

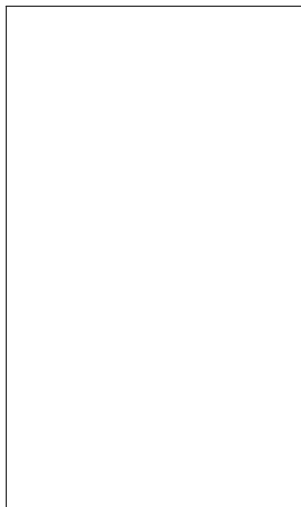
```
1 int power(int x, int n){
2     int res;
3     if ( n == 0)
4         res = 1;
5     else
6         res = x * power(x, n - 1);
7     return res;
8 }
```

Combien de mémoire est nécessaire pour exécuter `power(2,3)`? (sans optimisation)

- 4 octet par int
 - 3 entiers pour chaque appel de la fonction `power`
 - 4 appels récursifs
- 48 octets a réserver!

Exécution récursive avec allocation sur la pile

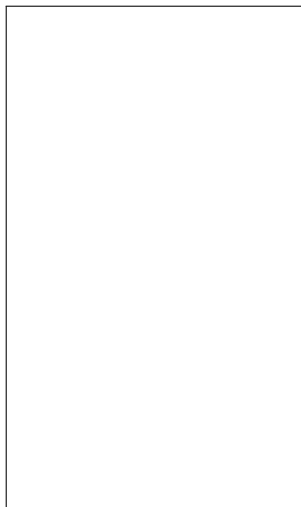
- $p(2,3)$
x = 2, n = 3, res = ?



(note: schéma à compléter)

Exécution récursive avec allocation sur la pile

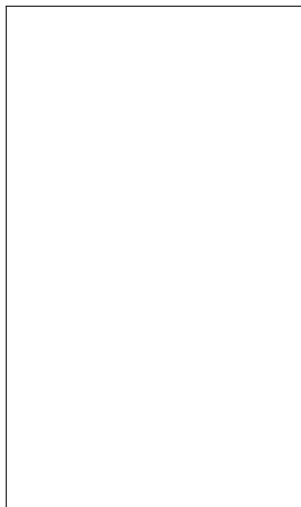
- $p(2,3)$
x = 2, n = 3, res = ?
- $p(2,2)$
x = 2, n = 2, res = ?



(note: schéma à compléter)

Exécution récursive avec allocation sur la pile

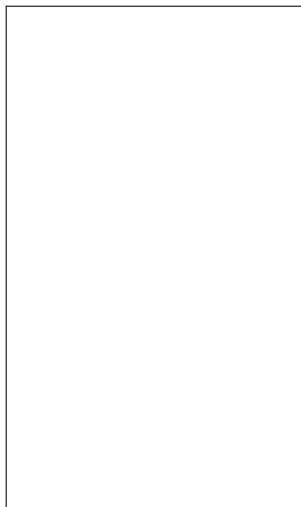
- $p(2,3)$
 $x = 2, n = 3, \text{res} = ?$
- $p(2,2)$
 $x = 2, n = 2, \text{res} = ?$
- $p(2,1)$
 $x = 2, n = 1, \text{res} = ?$



(note: schéma à compléter)

Exécution récursive avec allocation sur la pile

- $p(2,3)$
 $x = 2, n = 3, \text{res} = ?$
- $p(2,2)$
 $x = 2, n = 2, \text{res} = ?$
- $p(2,1)$
 $x = 2, n = 1, \text{res} = ?$
- $p(2,0)$
 $x = 2, n = 0, \text{res} = 1$



(note: schéma à compléter)

Pile et contexte d'exécution d'une fonction

La pile d'une fonction est déterminée par deux registres

- BP pour *Base Pointer*: début de la pile de la fonction en cours d'exécution
- SP pour *Stack Pointer*: fin de la pile de la fonction en cours d'exécution

Pile et contexte d'exécution d'une fonction

La pile d'une fonction est déterminée par deux registres

- BP pour *Base Pointer*: début de la pile de la fonction en cours d'exécution
- SP pour *Stack Pointer*: fin de la pile de la fonction en cours d'exécution

- L'espace entre BP et SP constitue le contexte d'exécution de la fonction
- Toutes les variables sont référencées de manière relative à BP
- Le SP et le BP de la fonction appelante sont restauré à la fin de l'exécution de la fonction

Allocation sur la pile

Code C

```
1 void f(){
2     int a = 2;
3     int b = 3;
4     int c = 4;
5     int d = a + b;
6     printf("%d %d %d\n", a, b, c, d);
7 }
```

Code Assembleur (simplifié)

```
1     pushq   %rbp           // sauvegarde le BP de la fonction appelante,
2     movq   %rsp, %rbp     // le SP devient le BP
3     subq   $16, %rsp      // réserve 4x sizeof(int)
4     movl   $2, -4(%rbp)   // écrit 2 dans a
5     movl   $3, -8(%rbp)   // écrit 3 dans b
6     movl   $4, -12(%rbp)  // écrit 4 dans c
7     movl   -4(%rbp), %edx  // copie a dans le registre EDX
8     movl   -8(%rbp), %eax  // copie b dans le registre EAX
9     addl   %edx, %eax     // ajoute le contenu de EDX et EAX
10    movl   %eax, -16(%rbp) // copie le résultat dans d
11    ...
12    call   printf@PLT
```

Erreur classique

```
1 int *f(){
2   int a = 2;
3   int b = 3;
4   int res = a + b;
5   return &res;
6 }
7 int main(){
8   int *r = f();
9   printf("%p %d\n", r);
10 }
```

Résultat ?

Erreur classique

```
1 int *f(){
2   int a = 2;
3   int b = 3;
4   int res = a + b;
5   return &res;
6 }
7 int main(){
8   int *r = f();
9   printf("%p %d\n", r);
10 }
```

Résultat ?

► Non défini!

- r pointe vers une adresse dans la pile de f
- la pile de f est libérée a la fin de l'exécution de f!

À l'exécution

```
1 #include <stdio.h>
2
3 int *f(){
4     int a = 2;
5     int b = 3;
6     int res = a + b;
7     return &res;
8 }
9
10 int g(){
11     printf("g écrase la pile\n");
12 }
13
14 int main(){
15     int *r = f();
16     // g();
17     printf("Res: %d\n", *r);
18 }
```

- Sans l'appel à g() dans main()

```
1 $./a.out
2 Res: 5
```

- Avec l'appel à g()

```
1 $./a.out
2 g écrase la pile
3 Res 0
```

Comment détecter ce bug

```
1 $ gcc demo-local-var.c
2 demo_local_var.c: In function 'f':
3 demo_local_var.c:7:9: warning: function returns address of local variable [-
    Wreturn-local-addr]
4     7 | return &res;
5     |           ~~~~
```


Comment détecter ce bug

```
1 $ gcc demo-local-var.c
2 demo_local_var.c: In function 'f':
3 demo_local_var.c:7:9: warning: function returns address of local variable [-
    Wreturn-local-addr]
4     7 | return &res;
5       |           ~~~~
```

 NE PAS IGNORER LES WARNINGS

Autre erreur classique

```
1 void broken(){
2     char T[4];
3     for(int i = 0; i <= 4; i++)
4         T[i] = 0;
5 }
6
7 int main(int argc, char **argv){
8     broken();
9 }
```

Résultat ?

Autre erreur classique

```
1 void broken(){
2     char T[4];
3     for(int i = 0; i <= 4; i++)
4         T[i] = 0;
5 }
6
7 int main(int argc, char **argv){
8     broken();
9 }
```

Résultat ?

► boucle infinie

Comment détecter ce bug

Avec un debugger

```
1 demo gdb 1
```

Autre erreur classique (2)

```
1 void broken(){
2     char *src = "abcd";
3     int len = strlen(src);
4     char *dest = malloc(len * sizeof(char));
5
6     for(int i = 0; i < len; i++)
7         dest[i] = src[i];
8
9     printf("%s",dest);
10
11 }
12
13
14 int main(int argc, char **argv){
15     broken();
16 }
```

Résultat ?

Autre erreur classique (2)

```
1 void broken(){
2     char *src = "abcd";
3     int len = strlen(src);
4     char *dest = malloc(len * sizeof(char));
5
6     for(int i = 0; i < len; i++)
7         dest[i] = src[i];
8
9     printf("%s",dest);
10
11 }
12
13
14 int main(int argc, char **argv){
15     broken();
16 }
```

Résultat ?

► probablement abcd

Comment détecter ce bug

Avec un debugger

```
1 demo valgrind
```