EPU - Informatique ROB4
Informatique Système
Gestion mémoire + divers

Miranda Coninx
Presented by
Ludovic Saint-Bauzel
ludovic.saint-bauzel@sorbonne-universite.fr

Sorbonne Université
Institut des Systèmes Intelligents et de Robotique (CNRS UMR 7222)

2023-2024

SORBONNE UNIVERSITÉ

ISIR
INSTITUT DES SYSTÈMES INTELLIGENTS ET DE ROBOTIQUE

Plan de ce cours

Gestion mémoire

- La mémoire virtuelle
- Anatomie de l'espace d'adressage virtuel d'un processus
- Fonctionnement de la pile
- mmap

Compléments sur le système de fichiers

- Utilisation de la fonction fstat() et stat()
- Exploration d'un répertoire
- Exercice : mini-ls

**** address-stack.c ****

```
int main(int argc, char* argv[])
{
        int i;
        printf("Local variable i has address %10p\n",&i);
        return 0;
}
```

Exécution

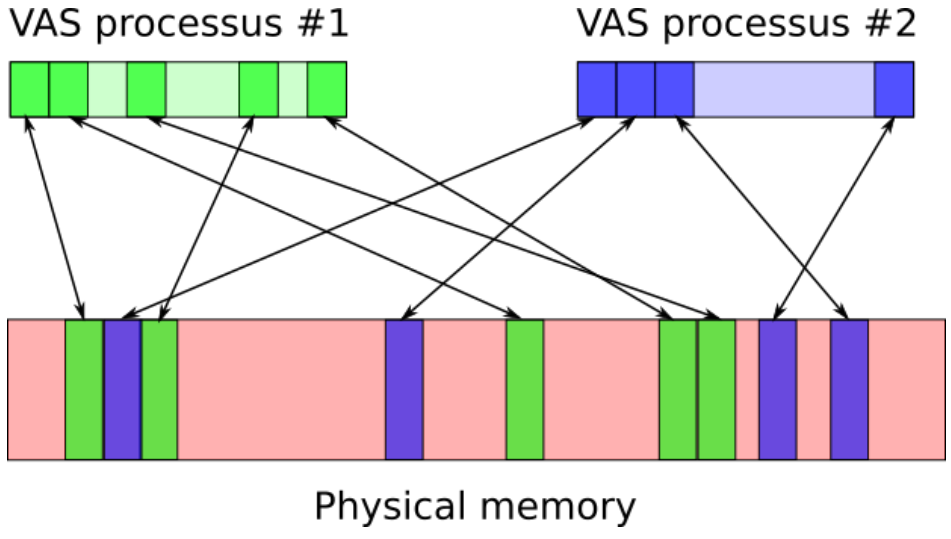
```
15:18 endy@demandred ~/mman% ./address-stack
Local variable i has address 0x7ffd727daf2c
```

- 0x7ffd727daf2c $= 140726524292908 \approx 128 \times 2^{40} - 10.2 \times 2^{30}$
- Presque 128 teraoctets de mémoire !
- Notre ordinateur n'a pas autant de RAM !

Possible grâce au mécanisme de la **mémoire virtuelle**

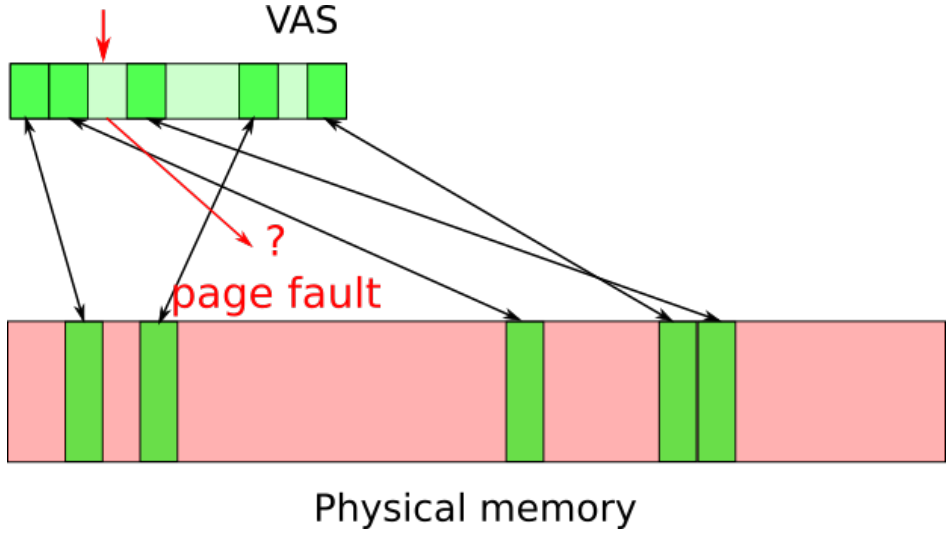
Principe

- Chaque processus possède un **espace d'adressage virtuel (VAS)**.
  - Cet espace est **propre à chaque processus**.
  - Taille : 128TB sur Linux 64 bits (2^{47} octets) (Pour les OS 32 bits : 4GB $= 2^{32}$ octets, le maximum adressable avec des adresses de 32 bits)
- Le VAS est subdivisé en **pages** (de 4kB sur la plupart des ordinateurs de bureau)
- 2^{35} pages par VAS (Linux 64 bits)
- Ces pages sont mise en correspondance avec des **cases** de la mémoire physique.
- ...mais seulement pour les pages effectivement utilisées par le processus (pour stocker le programme et les données)
- Le mapping entre mémoire physique et mémoire virtuelle est géré par la MMU (Memory Management Unit)



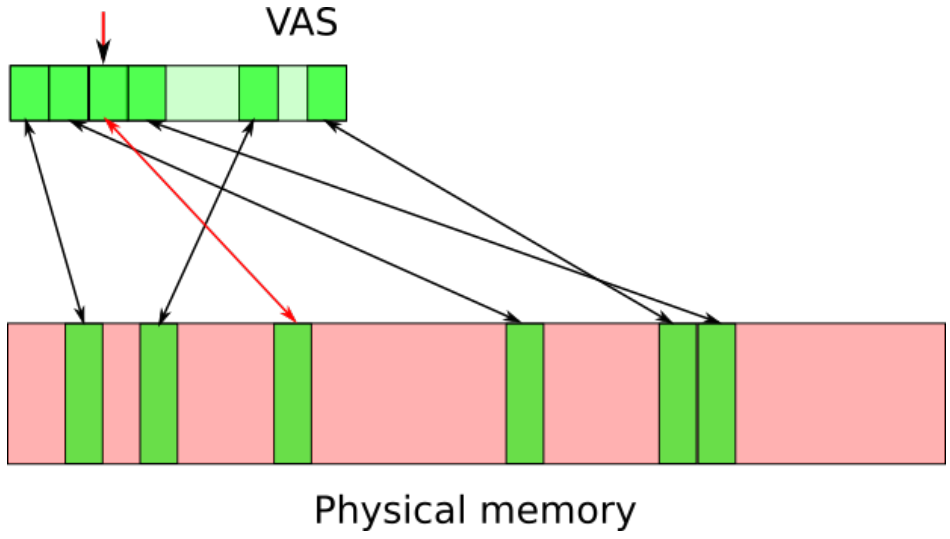
Gestion mémoire

- Si le processus veut accéder à une addresse dans une page non chargée, cela génère un **défaut de page** (page fault)
- Interruption logicielle, demande à la MMU de résoudre le problème (charger les données, allouer la mémoire)
- Si pas ássez de mémoire, on décharge une autre page (qui génèrera un défaut de page quand elle sera accédée)
- Quand charger et décharger quelle page pour minimiser les défauts de page et maximiser les performances ? Le problème de la gestion mémoire (se pose aussi pour le cache).
- Un exemple : algorithme LRU (Least Recently Used) : on supprime les pages les moins récemment accédées.



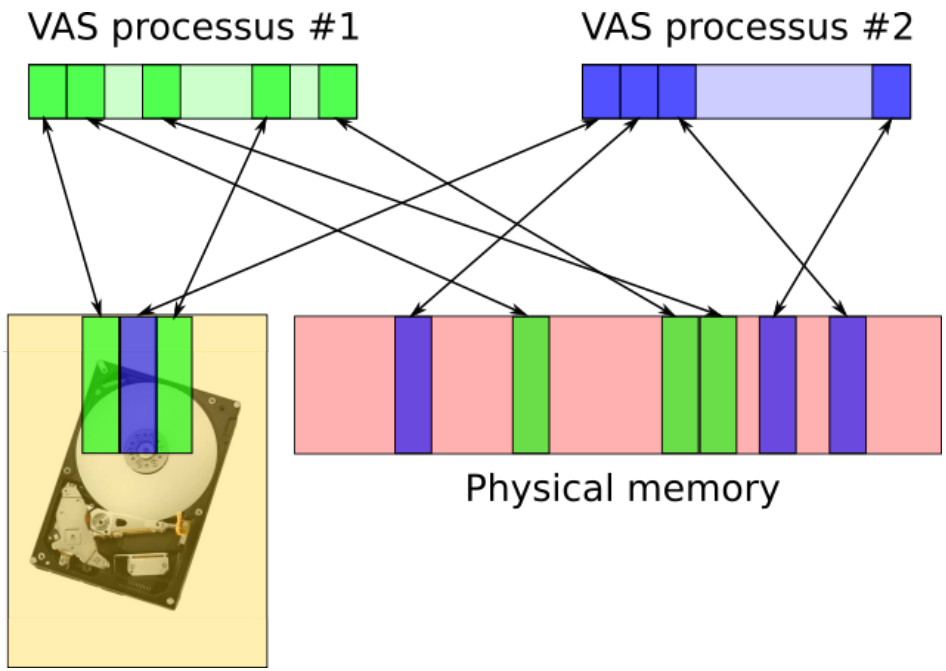
Gestion mémoire

- Si le processus veut accéder à une addresse dans une page non chargée, cela génère un **défaut de page** (page fault)
- Interruption logicielle, demande à la MMU de résoudre le problème (charger les données, allouer la mémoire)
- Si pas ássez de mémoire, on décharge une autre page (qui génèrera un défaut de page quand elle sera accédée)
- Quand charger et décharger quelle page pour minimiser les défauts de page et maximiser les performances ? Le problème de la gestion mémoire (se pose aussi pour le cache).
- Un exemple : algorithme LRU (Least Recently Used) : on supprime les pages les moins récemment accédées.



Adressage virtuel : avantages – 1

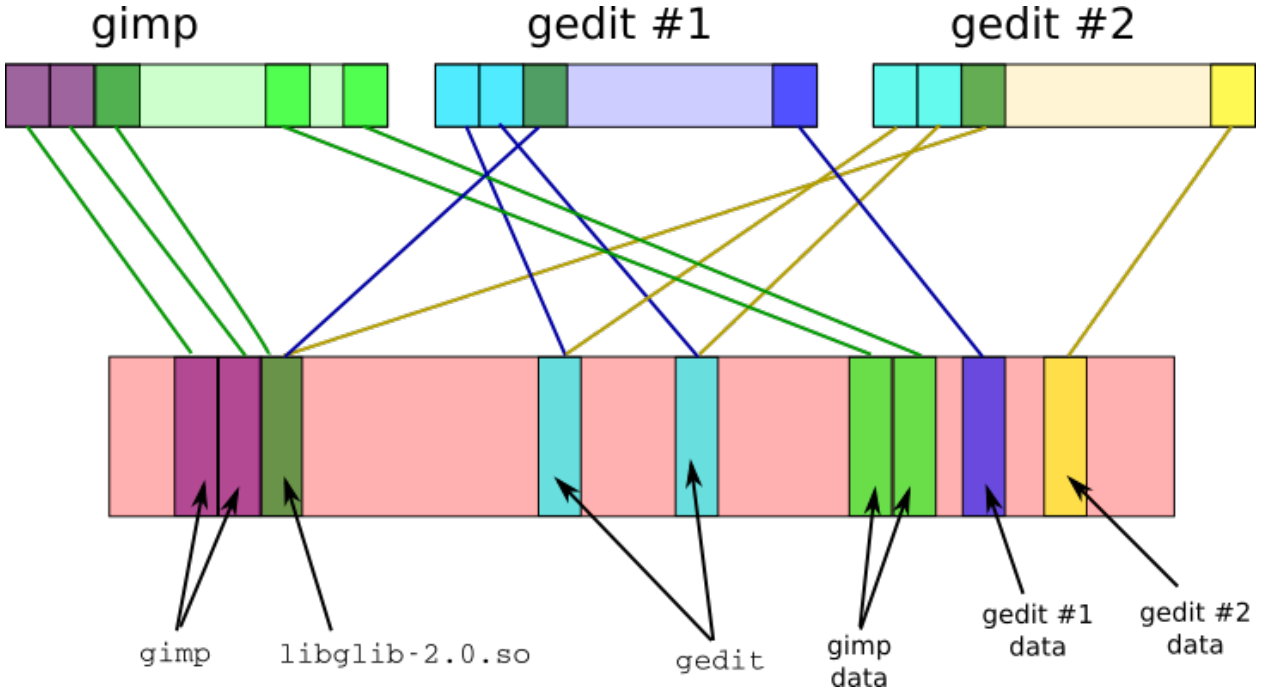
- Permet de **masquer la fragmentation mémoire**
  - Le processus accède à un espace mémoire continu, même si cela correspond à plusieurs plages de mmoire distinctes au sein de la mémoire physique
- Permet de mettre en place des **mécanismes de protection mémoire**
  - Chaque processus ne peut accéder quá la mémoire qui lui est dédiée
  - Chaque page a des permissions propres (comme un fichier) : lecture, écriture, exécution
  - Améliore la sécurité et la stabilité
- Permet de gérer des **ressources mémoire hétérogènes**
  - Si plus assez de RAM disponible, on peut temporairement stocker des pages mémoire sur le disque.
  - Dans une partition de swap (Linux) ou un fichier de swap (`pagefile.sys` sous Windows).
  - Géré par le noyau Linux et la MMU, de façon transparente pour le processus qui accède à la mémoire.



Adressage virtuel : avantages – 2

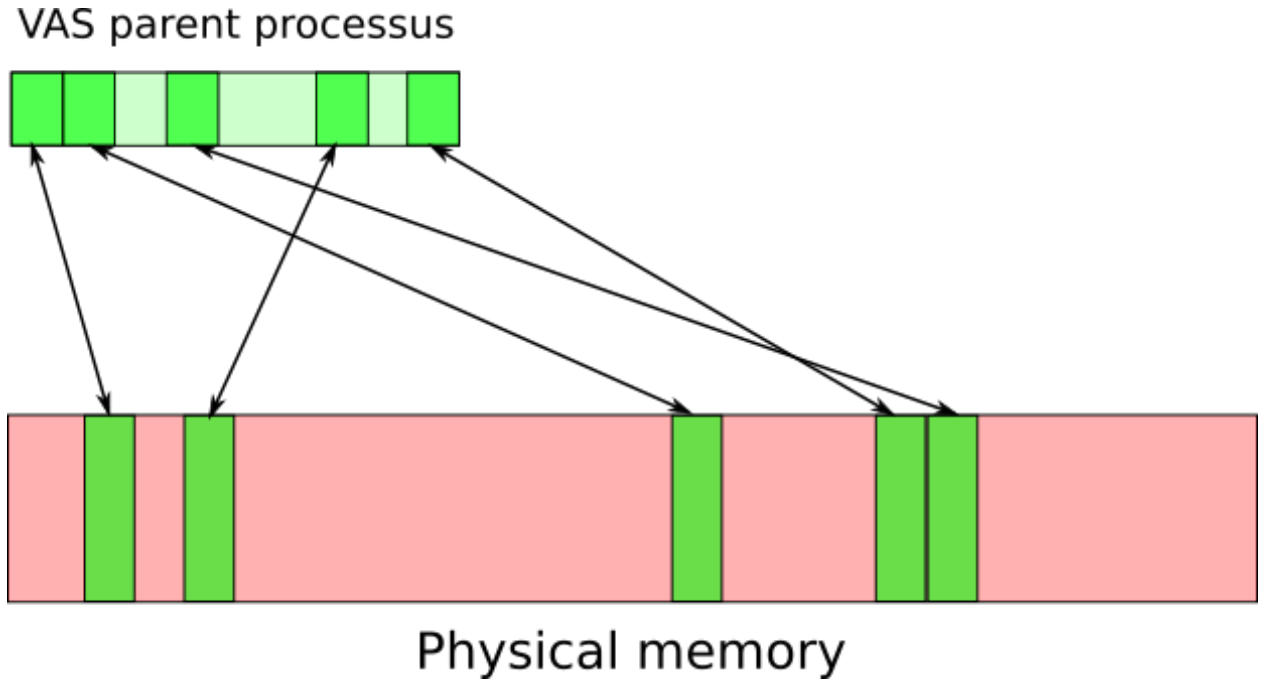
- Permet de partager les pages de code
  - Dans le cas où plusieurs processus executent le mème code
  - Egalement utilisé pour les bibliothèques partagèes (`.so`/`.dll`).
- Permet de projeter des fichiers en mémoire
  - Appel système `mmap`

Exemple : une instance de `gimp` et deux instances de `gedit` sont lancèes. `gimp` et `gedit` dépendent tous deux de la bibliothèque partagée `libglib-2.0.so`.



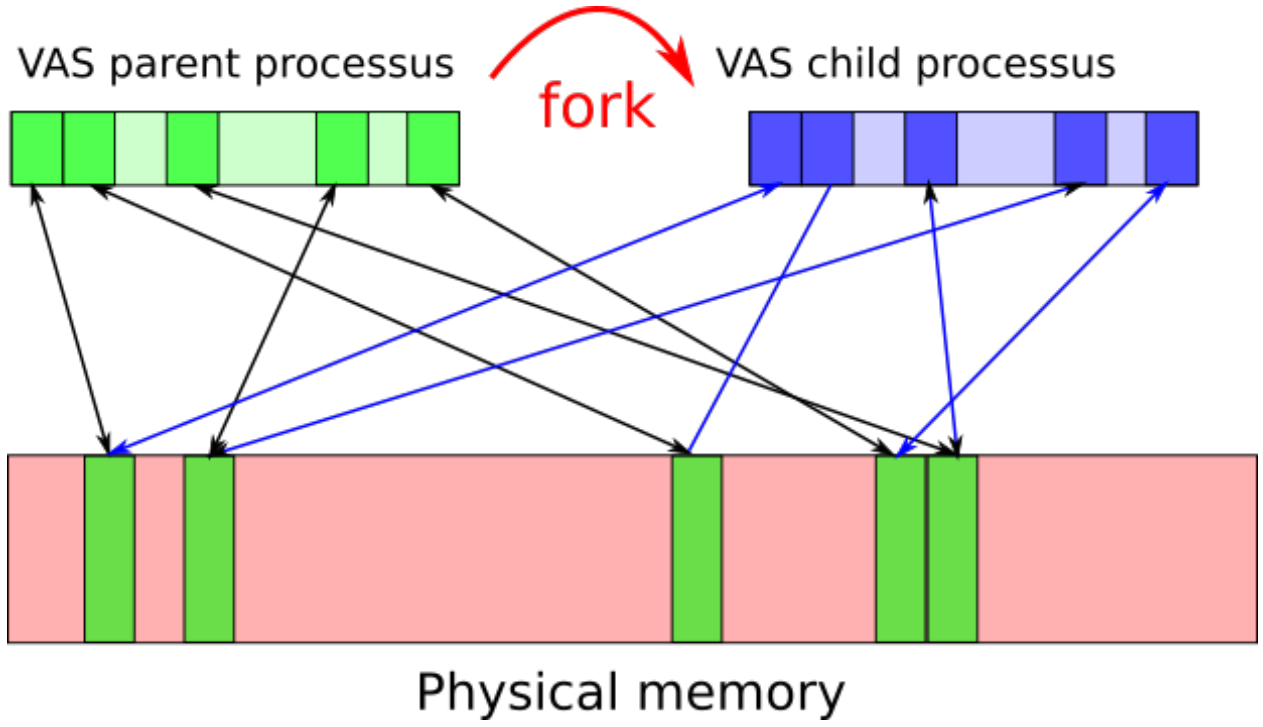
Adressage virtuel : avantages – 3

- Mécanisme de la copie en écriture
  - Après un `fork`, le processus enfant a un VAS propre...
  - ...mais il se réfère initialement au même espace mémoire que le père
  - Les pages mémoire sont copiées quand le processus veut les modifier
  - Mécanisme de **copie en écriture** (*copy-on-write*). Economise la mémoire.



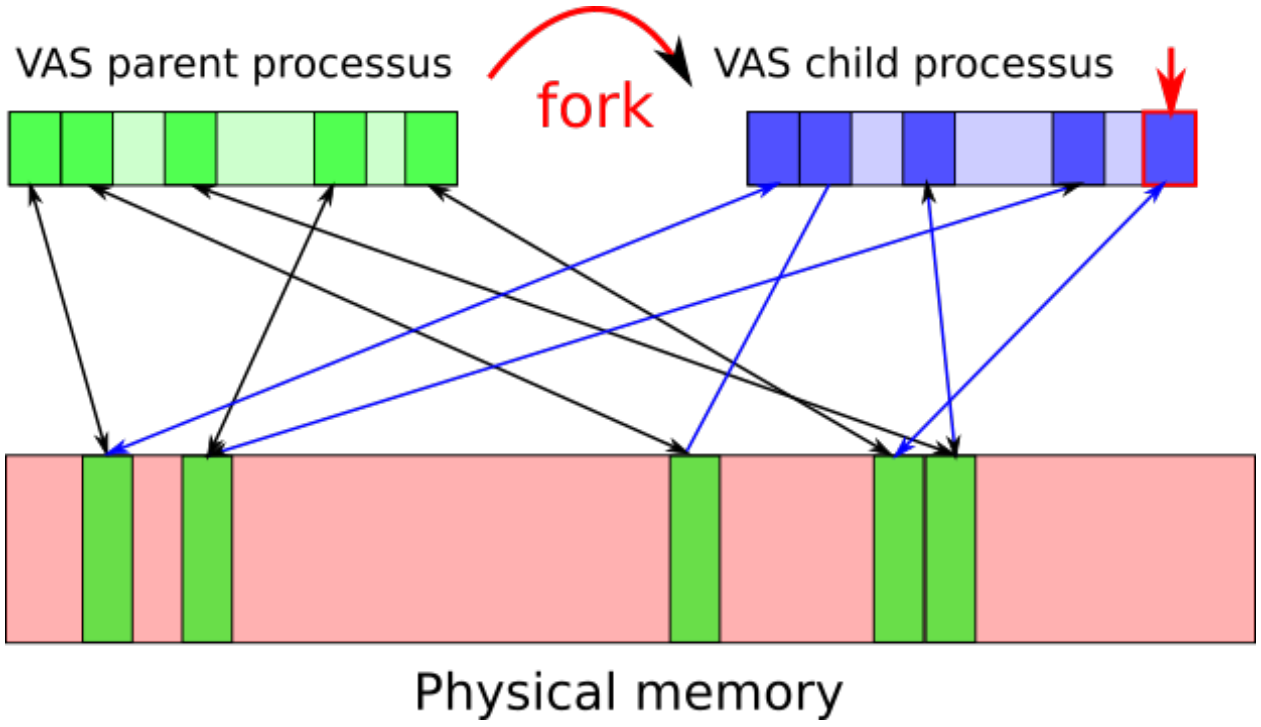
Adressage virtuel : avantages – 3

- Mécanisme de la copie en écriture
  - Après un `fork`, le processus enfant a un VAS propre...
  - ...mais il se réfère initialement au même espace mémoire que le père
  - Les pages mémoire sont copiées quand le processus veut les modifier
  - Mécanisme de **copie en écriture** (*copy-on-write*). Economise la mémoire.



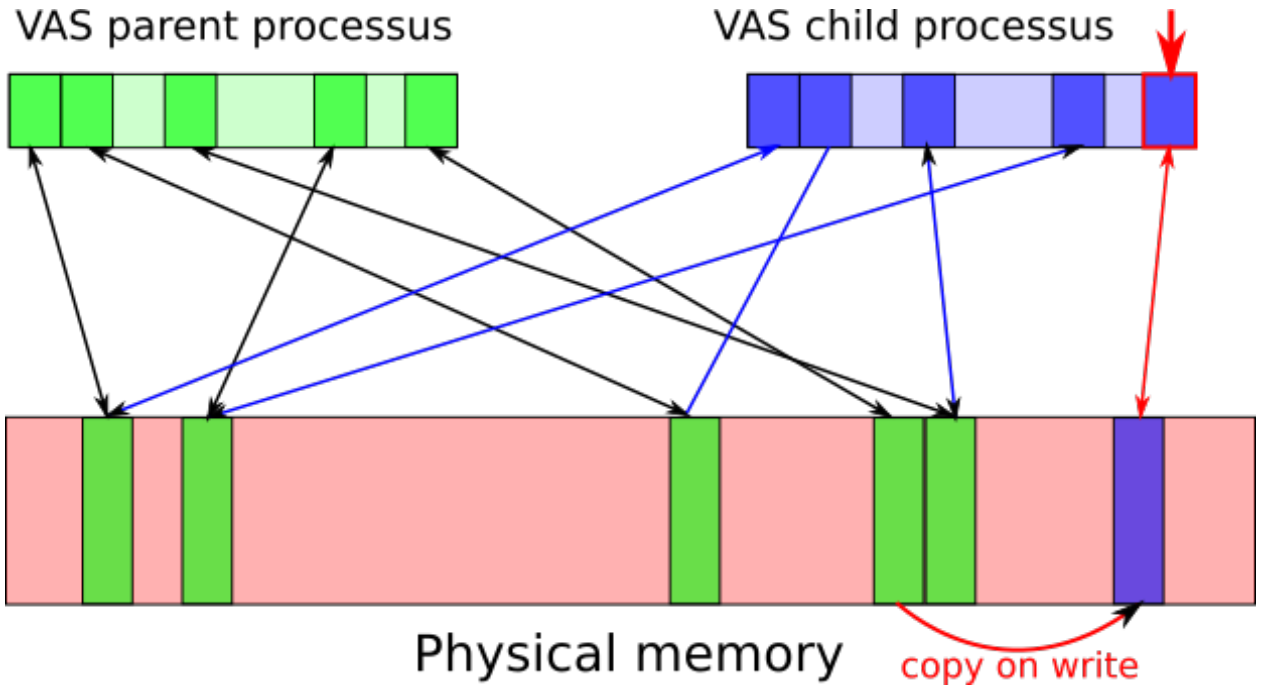
Adressage virtuel : avantages – 3

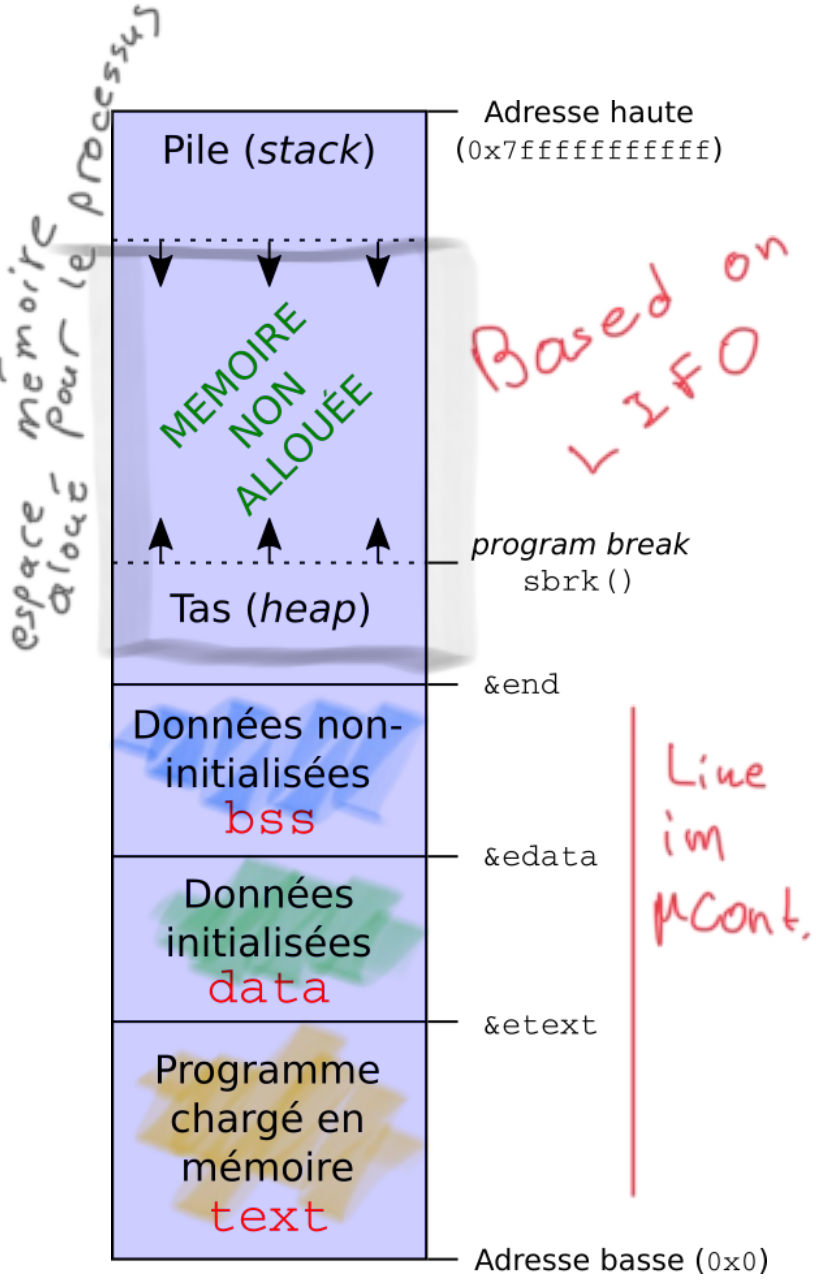
- Mécanisme de la copie en écriture
  - Après un `fork`, le processus enfant a un VAS propre...
  - ...mais il se réfère initialement au même espace mémoire que le père
  - Les pages mémoire sont copiées quand le processus veut les modifier
  - Mécanisme de **copie en écriture** (*copy-on-write*). Economise la mémoire.



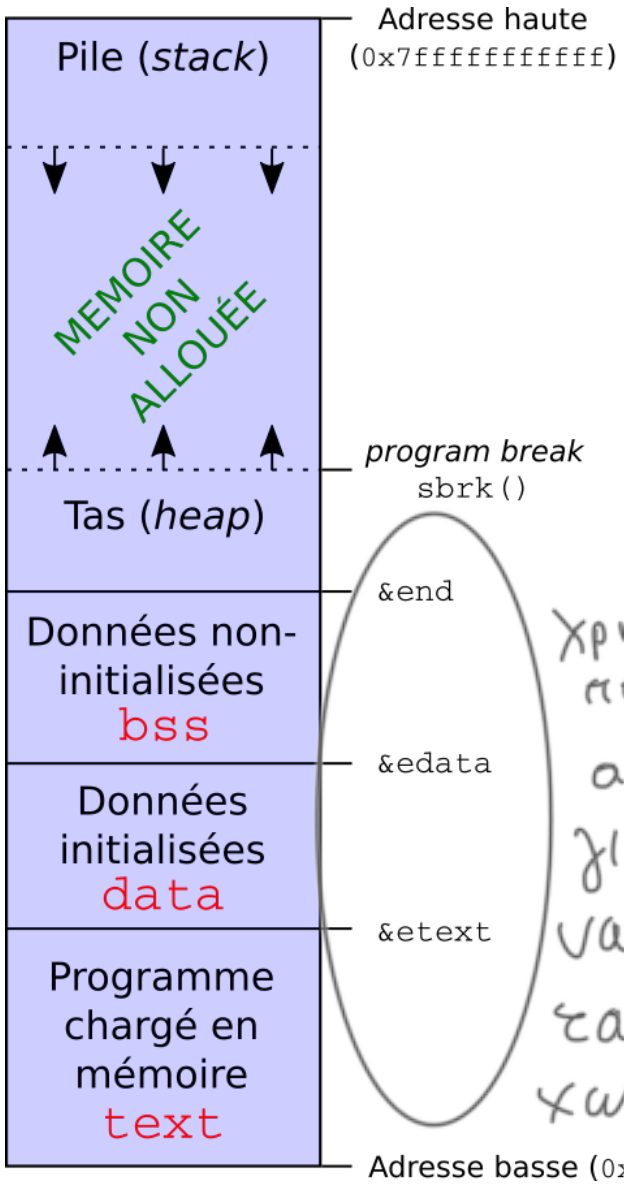
Adressage virtuel : avantages – 3

- Mécanisme de la copie en écriture
  - Après un `fork`, le processus enfant a un VAS propre...
  - ...mais il se réfère initialement au même espace mémoire que le père
  - Les pages mémoire sont copiées quand le processus veut les modifier
  - Mécanisme de **copie en écriture** (*copy-on-write*). Economise la mémoire.





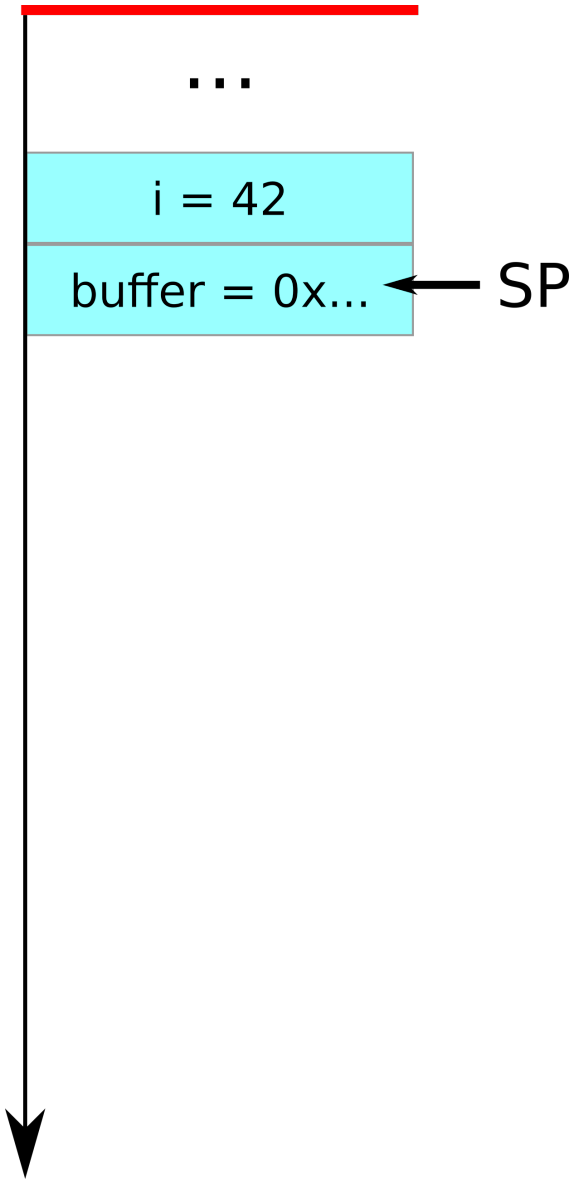
- L'espace d'adressage virtuel est divisé en plusieurs parties.
- La partie basse comprend des données chargées lors du chargement du processus, elles-mêmes subdivisées en **segments**.
  - Les limites de ces segments sont accessibles en inspectant l'adresse de variables globales `&etext`, `&edata`, `&end`.
- `text` correspond au **programme lui-même** (son code exécutable)
  - Les **pointeurs de fonctions** prennent des valeurs dans ce segment.
  - Géneralement en **lecture seule** sur les systèmes modernes : par défaut, un programme ne peut pas directement modifier la propre copie de lui-même chargée en mémoire.
- `data` correspond aux **données initialisées au chargement du programme**
  - Les **variables globales** (déclarées et initialisées en dehors de toute fonction)
  - Les **variables** `static` initialisées explicitement (`static int i = 42;`).
- `bss` correspond aux **données non-initialisées au chargement du programme**
  - Les **variables** `static` **non initialisées explicitement** (`static int i;`)
  - Cet espace mémoire entier est rempli de zéros au chargement du programme.



- La partie haute (au dessus de `&end`) comprend l'espace mémoire manipulé dynamiquement pendant l'exécution du processus.
- La **pile** (stack) contient les données directement liées à la fonction en cours d'exécution et celles qui l'ont appelée.
  - Les variables locales, arguments, valeurs de retour de la fonction en cours et de toutes les fonctions qui l'ont appelée.
  - Espace contigü et structuré
  - Part d'une adresse élevée et **croît vers le bas** (sur les PCs standard)
  - Accès implicite ; la pile croît quand des fonctions sont appelées et décroit avec `return`
  - Taille limitée (`ulimit -s`)
- Le **tas** (heap) contient les espaces mémoires allouées explicitement (par exemple par `malloc` en C)
  - Entre `&end` et une limite ajustable, le *program break*
    - Ajustable par `brk()` et `sbrk()`, utilisés en interne par `malloc()` et `free()`
  - Pas de structure particulière ; accès explicite à une adresse par un pointeur

segments.c

```
1
2  #define _GNU_SOURCE
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6  #include <sys/mman.h>
7
8
9  extern char etext, edata, end;
10
11 char foo[] = "Hello world\n";
12
13 int main(int argc, char* argv[])
14 {
15         void* x = malloc(100);
16         int i = 42;
17         static int mystatic;
18         char* anon = (char*)mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_ANON
19         printf ("end of program (etext) = %10p\n",&etext);
20         printf ("end of initialized data (edata) = %10p\n",&edata);
21         printf ("end of uninitialized data (end) = %10p\n",&end);
22         printf ("current program break (max heap) = %10p\n",sbrk(0));
23         printf ("=====\n");
24         printf ("Address of main() : %10p\n",&main);
25         printf ("Address of foo (global initialized) : %10p\n",&foo);
26         printf ("Address of mystatic (static variable) : %10p\n",&mystatic)
27         printf ("Address of i (on stack) : %10p\n",&i);
28         printf ("Value of pointer x (malloced; on heap) : %10p\n",x);
29         printf ("Value of pointer anon (mmapped) : %10p\n",anon);
30         free(x);
31         munmap(anon,4096);
```



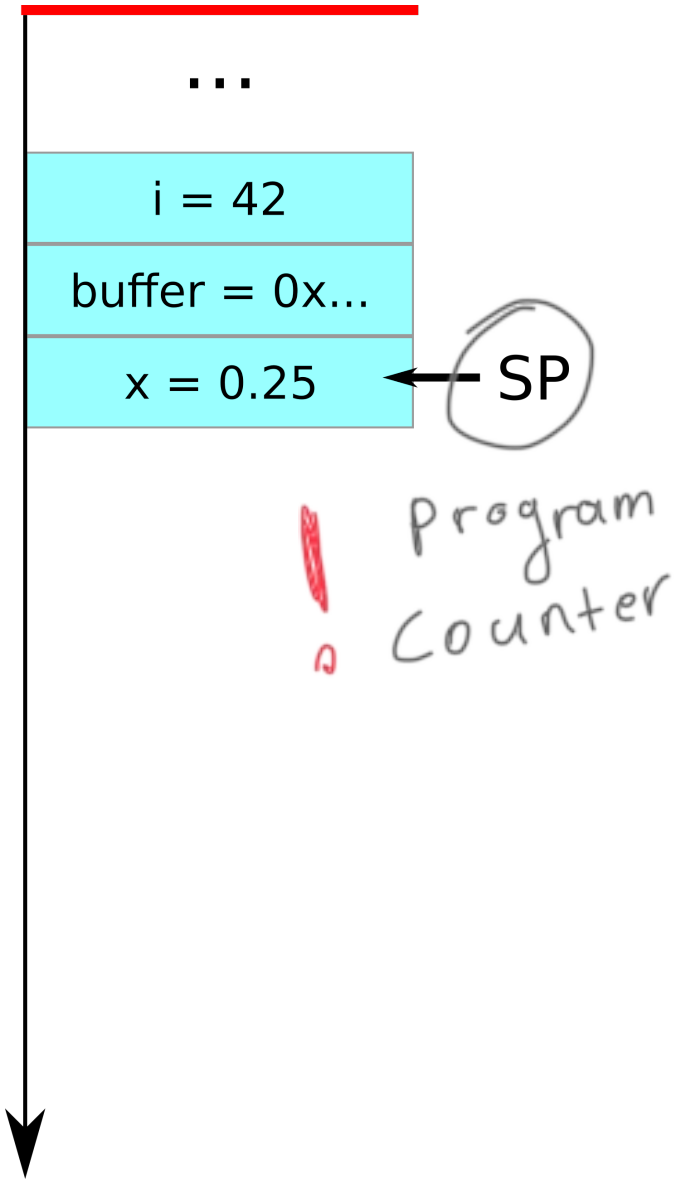
Exemple

```
 int foo(int val)
{
int j = 23;
return val+j;
}


int main(int argc, char* argv[])
{
int i = 42;
char * buffer = malloc(64*sizeof(char));
float x = 0.25;
int y = foo(i);
...
```

Fonctionnement de la pile

- La pile contient les variables locales de la fonction
- La **pointeur de pile** (SP, *stack pointer*) pointe vers le “sommet” de la pile.



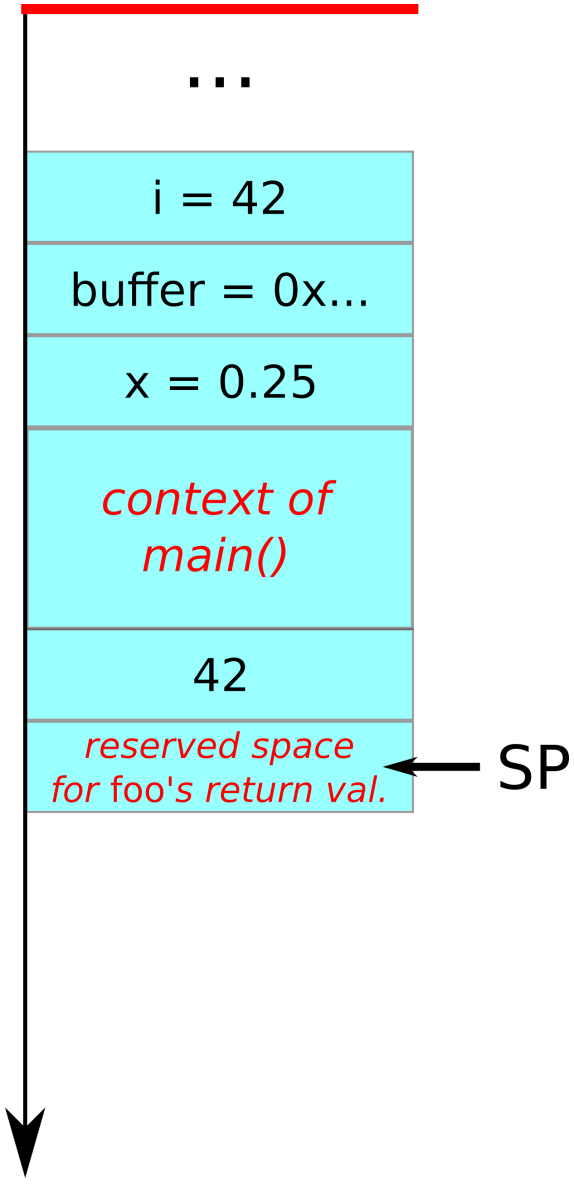
Exemple

```
 int foo(int val)
{
int j = 23;
return val+j;
}


int main(int argc, char* argv[])
{
int i = 42;
char * buffer = malloc(64*sizeof(char));
float x = 0.25;
int y = foo(i);
...
```

Fonctionnement de la pile

- Les variables locales sont ajoutées à la pile et le SP est mis à jour
- En interne, le programme compilé accède aux variables locales par leur position par rapport au SP.



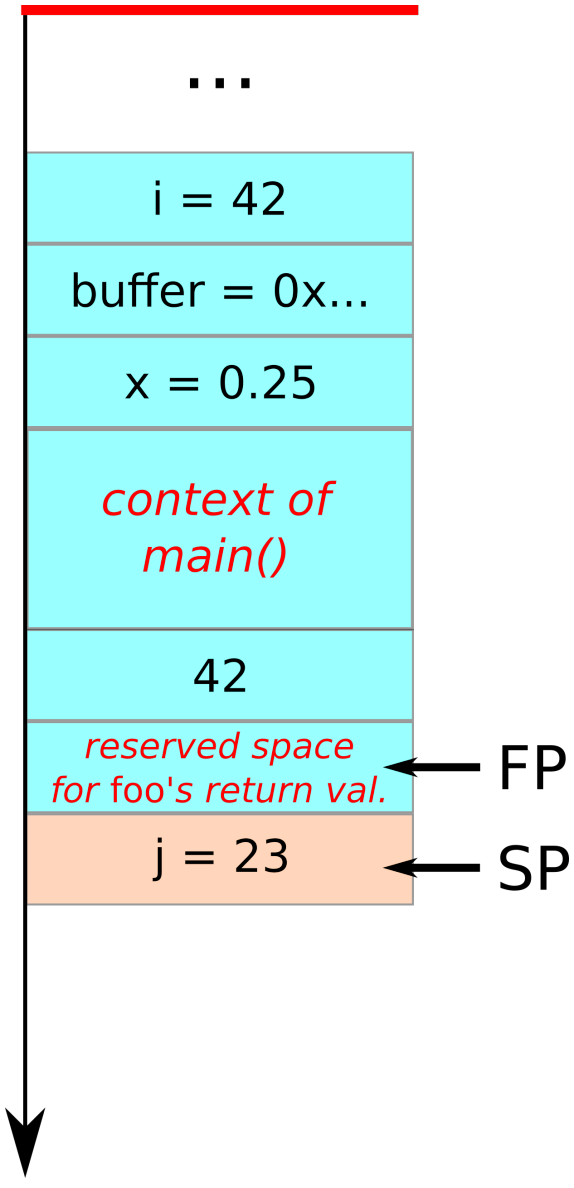
Exemple

```
 int foo(int val)
{
int j = 23;
return val+j;
}


int main(int argc, char* argv[])
{
int i = 42;
char * buffer = malloc(64*sizeof(char));
float x = 0.25;
int y = foo(i);
...
```

Fonctionnement de la pile

- Dans un appel de fonction, on empile :
  - Le **contexte de la fonction en cours** : ensemble des informations permettant de reprendre l'exécution à partir de ce point (état des registres, adresse de l'instruction suivante)
  - Les **arguments de la fonction appelée**
  - Un espace pour la **valeur de retour** de la fonction appelée



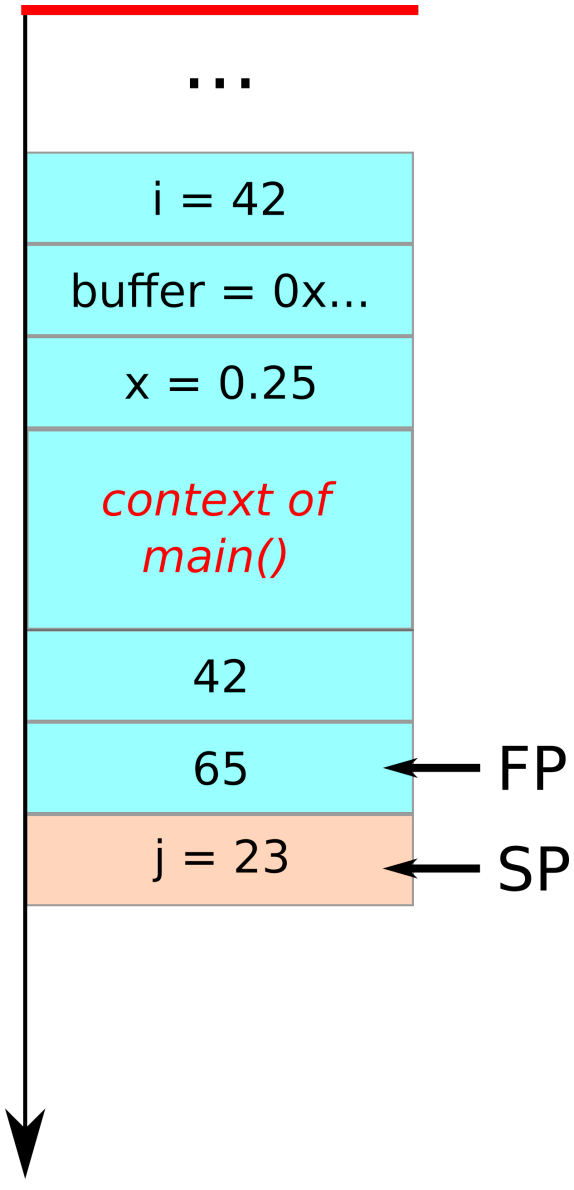
Exemple

```
 int foo(int val)
{
int j = 23;
return val+j;
}


int main(int argc, char* argv[])
{
int i = 42;
char * buffer = malloc(64*sizeof(char));
float x = 0.25;
int y = foo(i);
...
```

Fonctionnement de la pile

- Les variables locales de `foo` sont empilées comme celles de `main`
- Elles constituent la **stack frame** (“tranche de pile”) de `foo` (en jaune), au dessus de celle de `main` (en bleu)
- Le **frame pointer** (FP) pointe vers le sommet de la frame précédente



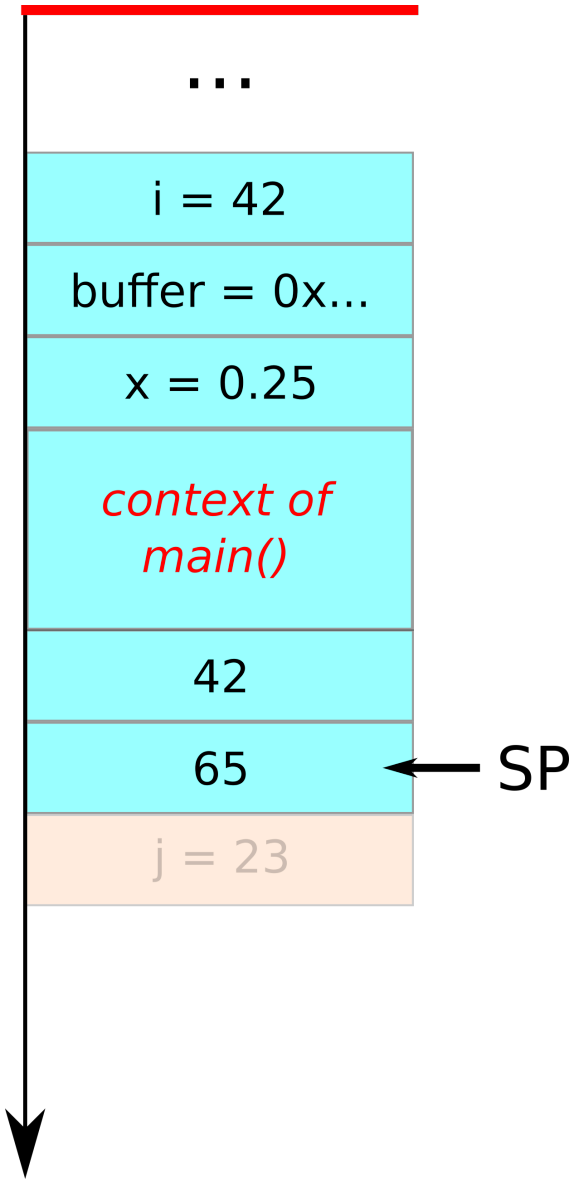
Exemple

```
 int foo(int val)
{
int j = 23;
return val+j;
}


int main(int argc, char* argv[])
{
int i = 42;
char * buffer = malloc(64*sizeof(char));
float x = 0.25;
int y = foo(i);
...
```

Fonctionnement de la pile

- Lors d'un `return` : la valeur de retour est mise à la valeur désirée



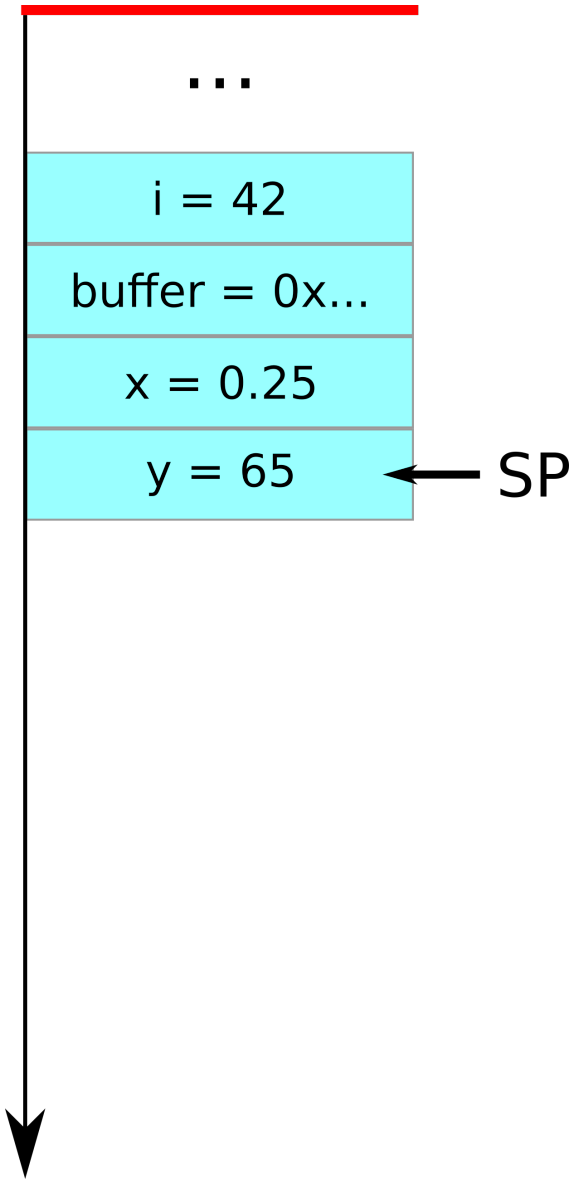
Exemple

```
 int foo(int val)
{
int j = 23;
return val+j;
}


int main(int argc, char* argv[])
{
int i = 42;
char * buffer = malloc(64*sizeof(char));
float x = 0.25;
int y = foo(i);
...
```

Fonctionnement de la pile

- Le *stack pointer* (SP) est fixé à l'adresse pointée par le *frame pointer* (FP).
- Les variables locales de `foo` sont perdues !



Exemple

```
 int foo(int val)
{
int j = 23;
return val+j;
}


int main(int argc, char* argv[])
{
int i = 42;
char * buffer = malloc(64*sizeof(char));
float x = 0.25;
int y = foo(i);
...
```

Fonctionnement de la pile

- La valeur de retour est récupérée et l'exécution continue...

L'appel système mmap

- L'appel `mmap` permet de **projeter en mémoire** le contenu d'un fichier
- Il renvoie un pointeur vers un espace mémoire ; lire ou écrire dans cet espace se traduit par une modification du fichier
- L'espace utilisé se trouve dans la zone non allouée de l'espace d'adressage virtuel (entre le *program break* et le sommet de la pile)
- `munmap` : supprime la projection
- Lecture/écriture bufferisée : le fichier est synchronisé lors de l'appel à `munmap`, ou avant en utilisant `msync`.

mmap

```
#include <sys/mman.h>}
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset
```

- `addr` : addresse a laquelle on *préfèrerait* que le mapping soit localisé (pas de garantie)
- `fd` : descripteur du fichier à projeter (obtenu par `open`)
- `length` octets du fichier à partir de `offset` sont projetés en mémoire
  - `offset` doit être un multiple de la taille d'une page mḿoire (4096 octets)
  - Le fichier doit **déjà avoir une taille suffisante** (impossible de l'étendre)
- `prot` : mode de protection du fichier
  - `PROT_READ` (lecture), `PROT_WRITE` (écriture), `PROT_EXEC` (éxecution)
  - Doit être cohérent avec le mode d'ouverture donné à `open` !
- `flags`
  - `MAP_SHARED` (ou `MAP_PRIVATE`)
  - `MAP_FILE` (ou `MAP_ANON`)
  - Détaillé plus loin
- Renvoie un pointeur vers la projection mémoire, ou -1 en cas d'échec

munmap

```
#include <sys/mman.h>}
void *munmap(void *addr, size_t length);}
```

- Supprime toutes les projections mémoire à partir de `addr` et sur `length`
- Synchronise aussi les fichiers projetés avec le contenu de la projection
- Renvoie 0 en cas de succès, -1 en cas d'erreur

msync

```
#include <sys/mman.h>}
int msync(void *addr, size_t length, int flags);
```

- Force la synchronisation de toutes les projections mémoire à partir de `addr` et sur `length` avec le(s) fichier(s)
- `flags` : `MS_ASYNC` (appel asynchrone, retourne immédiatement) ou `MS_SYNC` (appel synchrone, retourne quand les fichiers ont effectivement été mis à jour)
- Renvoie 0 en cas de succès, -1 en cas d'erreur

Exemple - mmap.c

```c
#define _GNU_SOURCE

#include <sys/mman.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
        char mystring[] = "hello world !";
        float pi = 3.1416;
        int fd = open("mystring", O_RDWR);
        int fd2 = open("myfloat", O_RDWR);
        char* mymap = (char*)mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                MAP_FILE|MAP_SHARED, fd, 0);
        float* mymap2 = (float*)mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                MAP_FILE|MAP_SHARED, fd2, 0);
        strcpy(mymap, mystring);
        *mymap2 = pi;
        munmap(mymap,4096);
        munmap(mymap2,4096);
        close(fd);
        close(fd2);
}
```

Une façon de plus de lire et écrire des fichiers – pour quoi faire ?

- `mmap` est assez complexe à utiliser
  - Demande un `open` préalable
  - Mode d'accès binaire
  - Pas de notion de curseur (on doit savoir où lire/écrire en utilisant un pointeur)
  - Pas possible d'étendre un fichier existant
- ...mais il est *extrêmement* performant

*** mmap-perf.c *** *** write-perf.c *** *** fwrite-perf.c ***

Performances

Remplir un fichier de 4096 octets avec la valeur binaire "42" (10000 fois)

- Avec `write` : 5.886s
- Avec `fwrite` : 0.337s
- Avec `mmap` : 0.107s

`mmap` anonyme

- Le flag `MAP_ANON` permet d'obtenir une projection anonyme, c'est à dire qui ne renvoie à aucun fichier.
- Le paramètre `fd` est ignoré
- La plage mémoire concernée est initialisée avec des 0
- Assez similaire à un `malloc`
  - D'ailleurs, `malloc` utilise parfois `mmap` en intene.

Particularité de `mmap` – mémoire partagée

- Le flag `MAP_SHARED` permet d'obenir une projection partagée entre un processus et ses enfants (sinon utiliser `MAP_PRIVATE`)
  - Possible car une partie de l'espace d'adressage est recouvert par le fichier projeté : les accès à cette zone ne renvoient plus à l'espace mémoire propre au processus.
- Permet d'utiliser `mmap` (en particulier avec `MAP_ANON`) pour la communication entre processus par mémoire partagée
- Très performant
- Demande de gérer les problèmes de concurrence et de synchronisation (par exemple avec des tubes)
- Exemple : **mmap-ipc.c**
- (Des mécanismes plus élaborés existent aussi : `shm_open()`, etc.)

Utilisation de la fonction fstat() et stat()

- Les attributs associés aux fichiers peuvent être récupérés par un appel aux primitives stat() et fstat() :
  `int stat(const char *ref, struct stat *infos);`
  `int fstat(const int desc, struct stat *infos);`
- Leur utilisation nécessite les inclusions suivantes :
  `#include <sys/types.h>`
  `#include <sys/stat.h>`
- Ces deux fonctions retournent les attributs associés à un fichier soit désigné par son nom (stat()) soit désigné par son descripteur (fstat()).
- Le type `struct stat` est défini dans `#include <sys/stat.h>` :

```
1  struct stat {
2      dev_t     st_dev;     /* ID of device containing file */
3      ino_t     st_ino;     /* inode number */
4      mode_t    st_mode;    /* protection */
5      nlink_t   st_nlink;   /* number of hard links */
6      uid_t     st_uid;     /* user ID of owner */
7      gid_t     st_gid;     /* group ID of owner */
8      dev_t     st_rdev;    /* device ID (if special file) */
9      off_t     st_size;    /* total size, in bytes */
10     blksize_t st_blksize; /* blocksize for file system I/O */
11     blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
12     time_t    st_atime;   /* time of last access */
13     time_t    st_mtime;   /* time of last modification */
14     time_t    st_ctime;   /* time of last status change */
15 };
```

Utilisation de la fonction fstat() et stat()

Le type de fichier codé dans le champ mode_t peut être obtenu en utilisant l'une des macros suivantes :

- S_ISBLK(infos->st_mode) , renvoie vrai si le fichier est un fichier spécial en mode bloc ;
- S_ISCHR(infos->st_mode) , renvoie vrai si le fichier est un fichier spécial en mode caractères ;
- S_ISDIR(infos->st_mode) , renvoie vrai si le fichier est un répertoire ;
- S_ISFIFO(infos->st_mode) , renvoie vrai si le fichier est un tube nommé ;
- S_ISREG(infos->st_mode) , renvoie vrai si le fichier est un fichier régulier ;
- S_ISLNK(infos->st_mode) , renvoie vrai si le fichier est lien symbolique ;
- S_ISSOCK(infos->st_mode) , renvoie vrai si le fichier est un socket ;

Errors :

- EACCES : search permission is denied in the path prefix (path resolution)
- ELOOP : too many symbolic links while traversing the path
- EOVERFLOW : path refers to a file whose size cannot be represented in the type off_t (e.g. application compiled on 32-bit platform without 64 bits flag enabled)

Exemple

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <time.h>

main() {
  struct stat info;
  char filepath[] = "test_stat.c";
  if (stat(filepath, &info) != 0)
    perror("stat() error");
  else {
    printf("\nstat(): information about %s:", filepath);
    printf("  inode:    %d\n",   (int) info.st_ino);
    printf(" dev id:    %d\n",   (int) info.st_dev);
    printf("   mode:    %08x\n",       info.st_mode);
    printf("  links:    %d\n",         info.st_nlink);
    printf("    uid:    %d\n",   (int) info.st_uid);
    printf("    gid:    %d\n",   (int) info.st_gid);
    printf("  atime:    %s",    ctime(&(info.st_atime)));
    printf("  mtime:    %s",    ctime(&(info.st_mtime)));
    printf("   size:    %d\n",   (int) info.st_size);
    printf("bl size:    %d\n",   (int) info.st_blksize);
    printf(" blocks:    %d\n",   (int) info.st_blocks);
    printf("regular:    %d\n",   (int) S_ISREG(info.st_mode));
    printf("    dir:    %d\n",   (int) S_ISDIR(info.st_mode));
  }
}
```

Exemple

```
$ ./test_stat

stat(): information about /home/vincent/un_truc.txt:
  inode:    1081492
 dev id:    2056
   mode:    000081a4
  links:    1
    uid:    1000
    gid:    1000
  atime:    Mon Nov  9 11:21:53 2009
  mtime:    Mon Nov  9 11:21:26 2009
bl size:    4096
 blocks:    16
regular:    1
    dir:    0
```

Un autre exemple :

**** test_stat_new.c ****

Exploration d'un répertoire

La consultation des répertoires est transparente vis-à-vis de l'implantation du système de fichiers au travers des quatre fonctions suivantes :

- DIR *opendir(**const char** *nom) renvoie un descripteur de répertoire lequel sera utilisé pour parcourir la liste des entrées une entrée à la fois.
- **struct** dirent *readdir(DIR *desc) renvoie l'entrée suivante du répertoire désigné par le descripteur, NULL si l'on est en fin de parcours.
- D'après la norme POSIX, une entrée de répertoire est au format suivant :

```
1 struct dirent {
2         ino_t  d_ino;              // inode de l'objet sur le disque
3         char   d_name[];     // nom de l'objet (fichier ou repertoire)
4 };
```

- L'utilisation de cette structure nécessite l'inclusion de **#include** <dirent.h> .
- **void** closedir(DIR *desc) referme le flot de lecture du répertoire, le descripteur n'est plus utilisable sauf pour le réouvrir.
- **void** rewinddir(DIR *desc) peut être utilisée afin de reprendre le parcours du répertoire désigné par descripteur au début en resynchronisant la lecture sur l'état courant du répertoire.

TEST : mini-ls

Write a program called `mini-ls` which prints the information about the content of given folder, exploiting :

- `opendir`
- `stat`
- `closedir`

By default, it takes the current folder (i.e. the one where the command is launched), otherwise a specified path :

```
$ ./minils
$ ./minils /home/icub/software/
```

mini-ls

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <stdio.h>
#include <time.h>

static int ls(const char* repertoire) {
  DIR           *descripteur;
  struct dirent *entree;
  struct stat info;
  descripteur = opendir(repertoire);
  if (descripteur == NULL) {
    fprintf(stderr,"Ouverture du repertoire %s impossible.\n",repertoire);
    return EXIT_FAILURE;
  }
  while ((entree = readdir(descripteur)) != NULL) {
    stat(entree->d_name, &info);
        if(!S_ISDIR(info.st_mode)){
            puts(entree->d_name);
                printf("    uid:   %d\n",   (int) info.st_uid);
                printf("    gid:   %d\n",   (int) info.st_gid);
                printf("  atime:   %s",   ctime(&(info.st_atime)));
                printf("  mtime:   %s",   ctime(&(info.st_mtime)));
                printf("bl size:   %d\n",   (int) info.st_blksize);
                printf(" blocks:   %d\n",   (int) info.st_blocks);
    }
  }
  closedir(descripteur);
  return EXIT_SUCCESS;
}
```

mini-ls

```
1
2  int main(int argc,char *argv[]) {
3    int i, retour;
4
5    if (argc==1)
6      retour = ls(".");
7    else {
8      retour = EXIT_SUCCESS;
9      for (i=1; i<argc; i++){
10       if (ls(argv[i]) != EXIT_SUCCESS){
11                 retour = EXIT_FAILURE;
12       }
13     }
14   }
15   exit(retour);
```

mini-ls

```
$ ./test_dir ..
..
    uid:   1000
    gid:   1000
  atime:   Mon Nov  9 12:13:05 2009
  mtime:   Mon Nov  9 12:10:56 2009
bl size:   4096
 blocks:   8
.
    uid:   1000
    gid:   1000
  atime:   Mon Nov  9 12:08:44 2009
  mtime:   Mon Nov  9 12:12:58 2009
bl size:   4096
 blocks:   8
cours6_progsys-slides.pdf
    uid:   1000
    gid:   1000
  atime:   Mon Nov  9 12:08:44 2009
  mtime:   Mon Nov  9 12:12:58 2009
bl size:   4096
 blocks:   8
```