La notion de processus

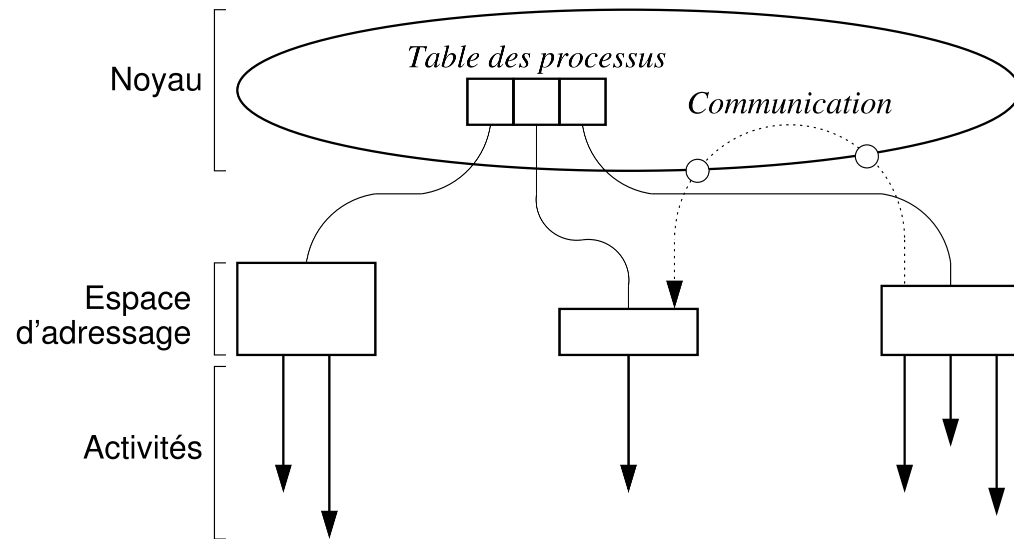
Description
Activation / Cycle de vie
Gestion de la mémoire

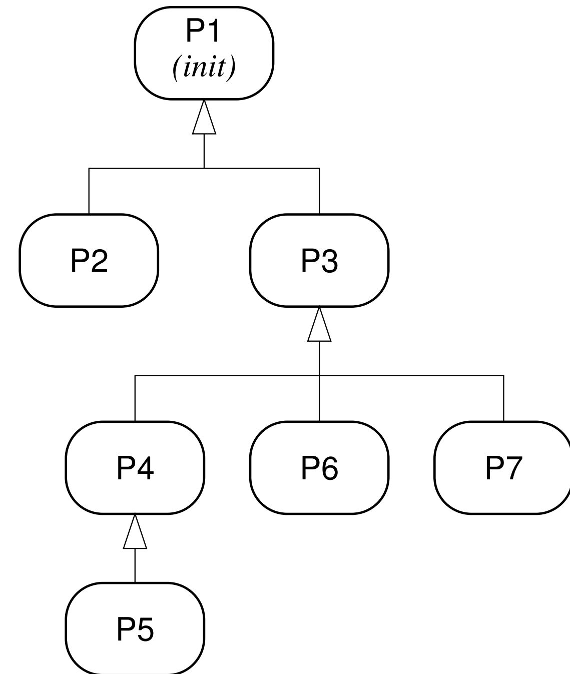
Fabrice Harrouet
École Nationale d'Ingénieurs de Brest
`harrouet@enib.fr`
`http://www.enib.fr/~harrouet/`

Description d'un processus

- **Notion centrale du système**
- **Un programme en cours d'exécution**
  - Le programme exécutable
  - Une activité (ou plusieurs)
  - Des données (plusieurs zones mémoire)
  - Un ensemble de registres
  - Des propriétés (environnement, droits . . .)
  - Des moyens de communication
- **Membre d'une hiérarchie**
  - Un parent
  - Zéro, un ou plusieurs enfants

Description d'un processus





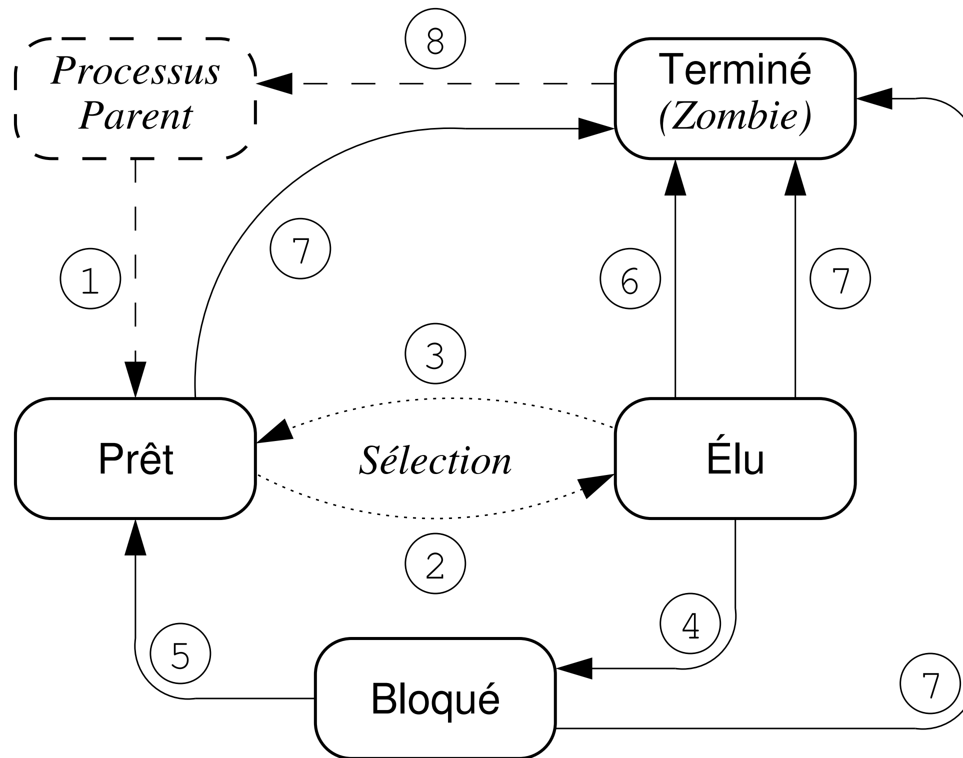
L'exécution d'un processus

- **Le programme exécutable**
  - Séquences d'instructions en *langage machine*
  - Calcul, tests, sauts ...
  - Des appels de procédures
  - Des gestionnaires de signaux
  - Des appels à des services de bibliothèque
  - Des appels *système*
- **Progression, évolution des registres, des données**
  - Chaque processus est indépendant
  - Progression en parallèle
  - Synchronisations explicites (communication)

Les propriétés d'un processus

- **Propriétés d'ordonnancement**
  - Un identifiant unique
  - Une priorité (un type de multi-tâches)
- **Réactions aux signaux**
  - Masques de signaux
  - Gestionnaires de signaux
- **L'identité de l'utilisateur**
  - Droits d'accès aux fichiers, aux périphériques ...
- **Variables d'environnement**
  - Données externes au programme
  - Influencer l'exécution

Cycle de vie d'un processus



1 : *Création*

2 : *Utilisation du processeur*

3 : *Un autre utilise le processeur*

4 : *Suspension ou attente de ressource*

5 : *Relance ou ressource disponible*

6 : *Terminaison normale*

7 : *Destruction*

8 : *Attente par le parent*

Identification d'un processus

▷ **Le** PID (*Process IDentifier*)

◇ Le type `pid_t` (≃ entier)

```
#include <sys/types.h>
```

◇ Récupérer l'identifiant (`man 2 getpid`)

```
#include <unistd.h>
pid_t getpid(void); // processus courant
pid_t getppid(void); // processus parent
```

◇ Toujours un résultat valide

Hiérarchie de processus

```
$ ps -A --forest
...
  805 ?        00:00:00 xdm
10079 ?        03:22:50  \_ X
10080 ?        00:00:00  \_ xdm
10953 ?        00:00:04      \_ fvwm2
10983 ?        00:00:01 rxvt
10986 pts/0    00:00:00  \_ bash
12833 pts/0    00:00:00      \_ vi
10988 ?        00:00:01 xosview
10991 ?        00:00:00 xclock
11003 ?        00:00:07 netscape-commun
11020 ?        00:00:00  \_ netscape-commun
11025 ?        00:00:00 plan
11037 ?        00:00:00 rxvt
11038 pts/1    00:00:00  \_ bash
11099 pts/1    00:00:00      \_ gv
11443 pts/1    00:00:01      |   \_ gs
12908 pts/1    00:00:00      \_ ps
11173 ?        00:00:00 rxvt
11174 pts/2    00:00:00  \_ bash
$
```

Création d'un processus

▷ **Appel système** `fork()` (`man 2 fork`)

◇
```
#include <unistd.h>
pid_t fork(void);
```

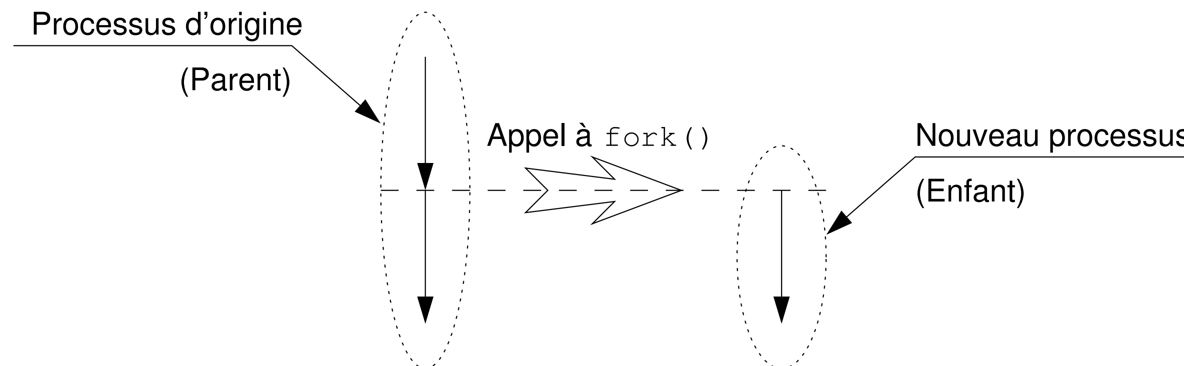
◇ Tous créés ainsi (sauf `init`)

▷ **Réplique quasi-identique du processus d'origine**

◇ Espace d'adressage (identique mais **indépendant**)

◇ Environnement, descripteurs de fichiers, signaux ...

▷ **Dédoublement du flot d'exécution**



Création d'un processus

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(void)
{
fprintf(stderr,"Begin %d\n",getpid());
pid_t result=fork();
switch(result)
  {
  case -1: /* Echec : pas assez de memoire ! */
    fprintf(stderr,"Pb with fork !\n");
    break;
  case 0: /*  Processus enfant : result = 0 */
    fprintf(stderr,"Child : %d (parent=%d)\n",getpid(),getppid());
    break;
  default: /* Processus parent : result = PID du processus fils */
    fprintf(stderr,"Parent : %d (child=%d)\n",getpid(),result);
  }
fprintf(stderr,"End %d\n",getpid());
return(0);
}
```

```
$ ./prog
Begin 13253
Parent : 13253 (child=13254)
Child : 13254 (parent=13253)
End 13254
End 13253
$
```

Création d'un processus

▷ **Deux causes d'échec possibles pour `fork()`**

◇ Mémoire insuffisante pour la table des processus

∘ `errno` vaut `ENOMEM`

∘ Erreur grave, système saturé !

◇ Mémoire insuffisante pour dupliquer le processus

∘ `errno` vaut `EAGAIN`

∘ La mémoire peut se libérer → recommencer

```
                                #include <errno.h>
                                ...
                                pid_t result;
pid_t result=fork();   ===>     do
                                  {
                                  result=fork();
                                  } while((result==-1)&&(errno==EAGAIN));
```

Terminaison d'un processus

▷ **Traitement à effectuer**

◇ Libérer les ressources du processus

◇ Signaler la terminaison au processus parent

- Envoi d'une valeur entière
- Généralement `0` signifie OK
- Constantes `EXIT_SUCCESS` et `EXIT_FAILURE` de `stdlib.h`

▷ **Retour de la fonction `main()`**

◇ Terminaison recommandée !

◇ Appel implicite de la fonction `exit()` avec le résultat

```
int main(void) { return 0; }
```

Terminaison d'un processus

- **Appel à la fonction** `exit()` (`man 3 exit`)
  - Depuis n'importe quel point du programme
  - `#include <stdlib.h>     void exit(int status);`
  - Fermeture des flux (synchronisation des tampons)
  - Appel des traitements de `atexit()`
  - Invocation de l'appel système `_exit()`
- **Appel système** `_exit()` (`man 2 _exit`)
  - Éviter d'utiliser directement cet appel !
  - `#include <unistd.h>     void _exit(int status);`
  - Fermeture des descripteurs de fichiers
  - Les enfants ont pour parent `init` (éviter l'état *zombie*)
  - Le parent recoit la valeur de retour, le processus disparait

Terminaison d'un processus

▷ **Les traitements de** `atexit()` (`man 3 atexit`)

- ◇ Enregistrer des traitements à déclencher à la terminaison
- ◇ Appel depuis **`exit()`** uniquement (et fin de **`main()`**)
- ◇
```
#include <stdlib.h>
int atexit(void (*function)(void));
```
- ◇ Uniquement des fonctions (utilisation de variables globales !)
- ◇ Appel dans l'ordre inverse d'enregistrement (pile)
- ◇ Résultat non nul → traitement non enregistré (raison ?)

Terminaison d'un processus

- **Réception d'un signal** (`man 2 kill` / `sigaction`)
  - Voir le cours Communication Sous *UNIX*
  - Envoi explicite ou suite à une erreur
  - Certains signaux provoquent la terminaison
    - Fermeture des descripteurs de fichiers
    - Les enfants ont pour parent `init` (état *zombie* temporaire)
    - Le parent est informé, le processus disparait
- **Appel à la fonction** `abort()` (`man 3 abort`)
  - Envoi du signal `SIGABRT` au processus courant
  - ```
    #include <stdlib.h>
    void abort(void);
    ```
  - Fermeture des flux (synchronisation des tampons)
    ```

Terminaison d'un processus

- **Vérification d'assertions** (`man 3 assert`)
  - S'assurer qu'une condition est bien vérifiée
  - ```
    #include <assert.h>
    void assert(int expression);
    ```
  - Si assertion non vérifiée
    - Affichage du fichier, de la ligne et de la condition
    - Appel de la fonction `abort()`
  - `assert()` est une *macro*
    - Sans effet si compilé avec `-DNDEBUG`
    - Utilisée seulement pour la mise au point
    ```

Attente d'un processus enfant

- **Appels systèmes `wait()` et `waitpid()`** (`man 2 wait`)
  - Attente du signal `SIGCHLD` d'un enfant vers le parent
  - ```
    #include <sys/types.h>
    #include <sys/wait.h>
    pid_t wait(int * status);
    pid_t waitpid(pid_t pid,int * status,int options);
    ```
  - Met à jour l'entier pointé par `status` (terminaison de l'enfant)
  - Options de `waitpid()` (`0` ou combinaison bit à bit)
    - `WNOHANG` : lecture immédiate, sans attente
    - `WUNTRACED` : enfants stoppés également
  - Retour : *PID* de l'enfant ou `0` si `WNOHANG` et pas d'enfant terminé
  - Causes d'erreurs : (retour vaut `-1`)
    - Enfant inconnu, mauvaises options, interruption (`EINTR` → relance)
    ```

Attente d'un processus enfant

▷ **Lecture du résultat**

- ◇ Macros pour décoder l'entier `status` (passer l'entier, pas son adresse)
- ◇ `WIFEXITED(status)` est vrai si terminaison normale de l'enfant
  - ◦ `WEXITSTATUS(status)` : valeur retournée par l'enfant
- ◇ `WIFSIGNALED(status)` est vrai si l'enfant est terminé par un signal
  - ◦ `WTERMSIG(status)` : signal reçu par l'enfant
- ◇ `WIFSTOPPED(status)` est vrai si l'enfant est bloqué
  - ◦ `WSTOPSIG(status)` : signal bloquant reçu par l'enfant
- ◇ Option `WUNTRACED` nécessaire pour `WIFSTOPPED` et `WSTOPSIG`

Attente d'un processus enfant

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>

int main(void)
{
fprintf(stderr,"Begin %d\n",getpid());
pid_t result;
do { result=fork(); } while((result==-1)&&(errno==EAGAIN));
switch(result)
  {
  case -1:
    fprintf(stderr,"Pb with fork !\n");
    break;
  case 0:
    {
    fprintf(stderr,"Child : %d (parent=%d)\n",getpid(),getppid());
    for(int i=0;i<100000000;i++) {} // attendre un peu
    // int * ptr=(int *)0; (*ptr)=1234; // Erreur de segmentation !
    } break;
```

Attente d'un processus enfant

```
  default:
    {
    fprintf(stderr,"Parent : %d (child=%d)\n",getpid(),result);
    int status;
    pid_t pid;
    do { pid=waitpid(result,&status,0); } while((pid==-1)&&(errno==EINTR));
    fprintf(stderr,"After waitpid() : %d\n",pid);
    if(WIFEXITED(status))
      fprintf(stderr,"  exit %d\n",WEXITSTATUS(status));
    else if(WIFSIGNALED(status))
      fprintf(stderr,"  signal %d\n",WTERMSIG(status));
    else
      fprintf(stderr,"  termination ?\n");
    } break;
  }
fprintf(stderr,"End %d\n",getpid());
return(0);
}
```

Attente d'un processus enfant

```
Sans l'erreur volontaire        Avec l'erreur volontaire
------------------------        ------------------------

Begin 16597                     Begin 16607
Parent : 16597 (child=16598)    Parent : 16607 (child=16608)
Child : 16598 (parent=16597)    Child : 16608 (parent=16607)
End 16598                       After waitpid() : 16608
After waitpid() : 16598           signal 11
  exit 0                        End 16607
End 16597
```

Créer (et attendre) des processus enfants

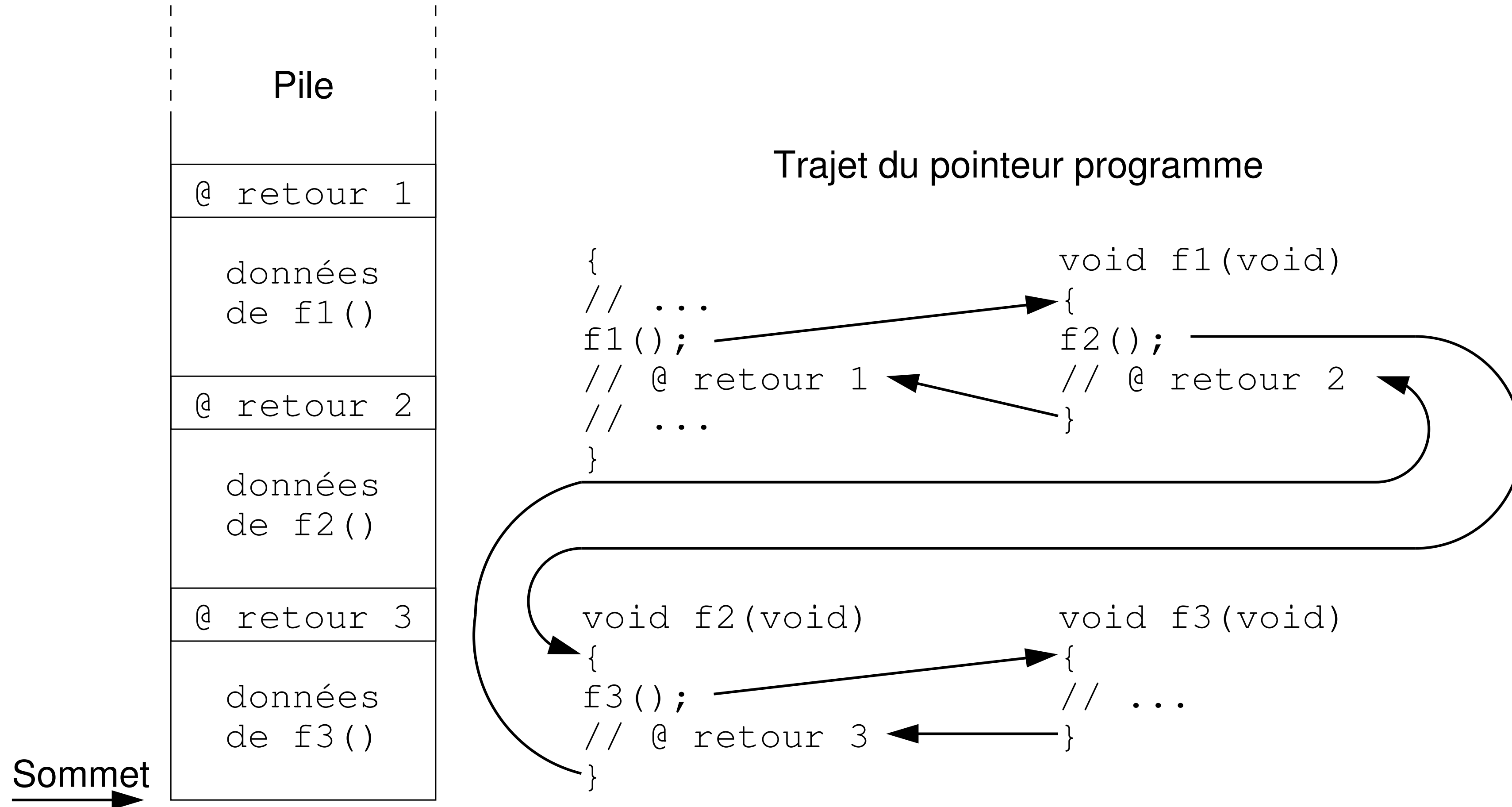
- **Premier cas : les enfants rendent un service au parent**
  - Attente des enfants, par nécessité, avec `wait()` ou `waitpid()`
    - Poursuite du traitement quand les enfants sont terminés
    - Traitement annexe envisageable avec `WNOHANG`
- **Deuxième cas : les enfants poursuivent seuls le traitement**
  - Le parent **doit** attendre les enfants pour qu'ils quittent l'état *zombie*
    - Utilisation périodique de `waitpid()` avec `WNOHANG`
    - Appel à `wait()` à l'intérieur du gestionnaire de signal de `SIGCHLD` (voir le cours Communication Sous *UNIX*)

Les sauts non locaux

▷ **Principe**

- ◇ Exécution normale : imbrication de fonctions
  - ◦ Empilement de l'adresse de retour
  - ◦ Saut à l'adresse de la fonction
  - ◦ Dépilement de l'adresse de retour
  - ◦ Saut à cette adresse
- ◇ Saut local : à l'intérieur d'une même fonction
  - ◦ Définir un label : `myLabel:`
  - ◦ Saut inconditionnel : `goto myLabel;`
- ◇ Saut non local : sortir instantanément de plusieurs imbrications
  - ◦ Sauver le contexte courant (sommet de pile, registres ...)
  - ◦ Poursuivre les imbrications
  - ◦ Restaurer le contexte sauvé

Les sauts non locaux



Les sauts non locaux

- **Le type** `sigjmp_buf` (`man 3 sigsetjmp`)
  - Type opaque pour sauvegarder un contexte d'exécution
  - C'est un tableau → implicitement manipulé par son adresse
- **La fonction** `sigsetjmp()` (`man 3 sigsetjmp`)
  - ```
    #include <setjmp.h>
    int sigsetjmp(sigjmp_buf ctx,int savesigs);
    ```
  - Sauve le contexte courant dans `ctx`
  - Masque des signaux sauvé si `savesigs` est non nul
    - C'est très généralement le cas
    - Voir le cours Communication Sous *UNIX*
  - Retour : `0` lors de l'invocation, non nul après `siglongjmp()`
    ```

Les sauts non locaux

▷ **La fonction** `siglongjmp()` (`man 3 siglongjmp`)

- ◇
```
#include <setjmp.h>
void siglongjmp(sigjmp_buf ctx,int value);
```
- ◇ Restaure le contexte `ctx`
- ◇ L'exécution reprend au `sigsetjmp()` correspondant
  - ∘ Si `value` vaut `0` → remplacé par `1`
  - ∘ Le retour de `sigsetjmp()` vaut `value`
- ◇ Utile pour signaler une erreur
- ◇ Ce n'est pas une transaction !
  - ∘ Les traitements effectués ne sont pas annulés
  - ∘ Les allocations dynamiques ne sont pas libérées
  - ∘ Les destructeurs *C++* ne sont pas appelés
→ utiliser de préférence `try`/`throw`/`catch`

Les sauts non locaux

```
#include <setjmp.h>
#include <stdio.h>
sigjmp_buf context;
void f(int i)
{
fprintf(stderr,"  Begin f(%d)\n",i);
if(i>=2) siglongjmp(context,i);
fprintf(stderr,"  End f(%d)\n",i);
}
void g(int n)
{
fprintf(stderr,"Begin g(%d)\n",n);
for(int i=0;i<n;i++) f(i);
fprintf(stderr,"End g(%d)\n",n);
}

int main(int argc,char ** /* argv */)
{
int result=sigsetjmp(context,1);
if(!result) { g(argc); fprintf(stderr,"OK\n"); }
else fprintf(stderr,"PB ! result=%d\n",result);
return(0);
}
```

```
$ ./prog
Begin g(1)
  Begin f(0)
  End f(0)
End g(1)
OK
$ ./prog a
Begin g(2)
  Begin f(0)
  End f(0)
  Begin f(1)
  End f(1)
End g(2)
OK
$ ./prog a b
Begin g(3)
  Begin f(0)
  End f(0)
  Begin f(1)
  End f(1)
  Begin f(2)
PB ! result=2
$
```

Les droits d'un processus

▷ **Identifier l'utilisateur associé au processus**

◇ Droits associés à l'utilisateur qui lance le processus

◇ Droits associés au programme lui-même

◇ Généralement différents

◇ Identiques si bit *"Set-UID"* positionné

```
$ ls -l prog*
-rwxr-xr-x    1 harrouet li2         12925 Jul 18 13:31 prog1
-rwxr-xr-x    1 harrouet li2         13130 Jul 18 13:32 prog2
$ su
Password:
# chown root.root prog2
# ls -l prog*
-rwxr-xr-x    1 harrouet li2         12925 Jul 18 13:31 prog1
-rwxr-xr-x    1 root     root        13130 Jul 18 13:32 prog2
# chmod +s prog2
# ls -l prog*
-rwxr-xr-x    1 harrouet li2         12925 Jul 18 13:31 prog1
-rwsr-sr-x    1 root     root        13130 Jul 18 13:32 prog2
```

Les droits d'un processus

▷ **L'`UID` et le `GID`** (*User/Group IDentifier*)

◇ Les types `uid_t` et `gid_t` (≃ entier)

```
#include <sys/types.h>
```

◇ Récupérer l'identifiant (`man 2 getuid` / `getgid`)

```
#include <unistd.h>
uid_t getuid(void); // utilisateur réel
uid_t geteuid(void); // utilisateur effectif
gid_t getgid(void); // groupe réel
gid_t getegid(void); // groupe effectif
```

◇ Toujours un résultat valide

◇ Réel : celui qui a lancé le processus

◇ Effectif : propriétaire du programme si bit *"Set-UID"* positionné

◇ Utilisateur/groupe **effectif** → droits du processus

Les droits d'un processus

▷ **Changer les droits du processus**
(`man 2 setuid` / `seteuid` / `setgid` / `setegid`)

◇
```
#include <unistd.h>
int setuid(uid_t uid); // utilisateur réel
int seteuid(uid_t euid); // utilisateur effectif
int setgid(gid_t gid); // groupe réel
int setegid(gid_t egid); // groupe effectif
```

◇ Retour : `0` si OK, `-1` si non autorisé

◇ Possibilité de perdre des droits

- Ne pas abuser des droits temporairement inutiles

◇ Impossibilité d'en gagner !

- Uniquement retrouver ses droits initiaux

Les droits d'un processus

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

int main(void)
{
fprintf(stderr,
        "Real=%d Effective=%d\n",
        getuid(),geteuid());
return(0);
}
```

```
$ ls -l prog
-rwxr-xr-x  1 harrouet li2   12229 Jul 18 18:09 prog
$ ./prog
Real=666 Effective=666
$ su
Password:
# ./prog
Real=0 Effective=0
# chown root.root prog
# chmod +s prog
# ls -l prog
-rwsr-sr-x  1 root     root  12229 Jul 18 18:09 prog
# ./prog
Real=0 Effective=0
# exit
$ ./prog
Real=666 Effective=0
```

Les droits d'un processus

▷ **Utilisation typique des droits**

- ◇ Donner au programme les droits `root`
  (accès aux périphériques, certains fichiers . . .)
  ```
  # chown root.root prog
  # chmod +s prog
  ```

▷ **Danger :**

- ◇ Inutile pour la majorité des traitements !
- ◇ Erreur de programmation → faille de sécurité
- ◇ Lancement d'autres traitements en tant que `root` (*shell* !)

▷ **Solution :**

- ◇ Abandonner les droits dès le début du programme
- ◇ Ne les reprendre que de manière très localisée
  (accès effectif aux périphériques, aux fichiers . . .)

Les droits d'un processus

```
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
uid_t effective=geteuid(); // Sauvegarder les droits initiaux
seteiud(getuid()); // Adandonner les droits au plus tot
/*
 * ...
 * Traitements ne necessitant pas de droits particuliers
 * ...
 */
seteiud(effective); // Reprendre temporairement les droits
/****  Traitement necessitant les droits accordes initialement au programme  ****/
seteiud(getuid()); // Abandonner les droits des qu'ils ne sont plus necessaires
/*
 * ...
 * Traitements ne necessitant pas de droits particuliers
 * ...
 */
return(0);
}
```

Accès à l'environnement

▷ **Un ensemble de variables d'environnement**

- Un nom et une valeur (chaîne de caractères)
- Transmises aux enfants (modifications éventuelles)
- Pas de signification *a priori* mais certaines conventions
  - `PATH`, `LD_LIBRARY_PATH`, `LD_PRELOAD`, `SHELL`, `PS1`, `TERM` ...
- Lecture depuis un programme
  - Paramétrer, personnaliser le programme
  - Généralement positionnées dans le `.profile`

Accès à l'environnement

▷ **L'environnement comme argument de `main()`**

◇ Solution non recommandée par *Posix* !

◇ `int main(int argc,char ** argv,char ** envp)`

◇ Structure de `envp` similaire à `argv`

∘ Tableau de chaînes au format `NOM=VALEUR`

∘ Terminé par un chaîne nulle

▷ **L'environnement comme variable globale**

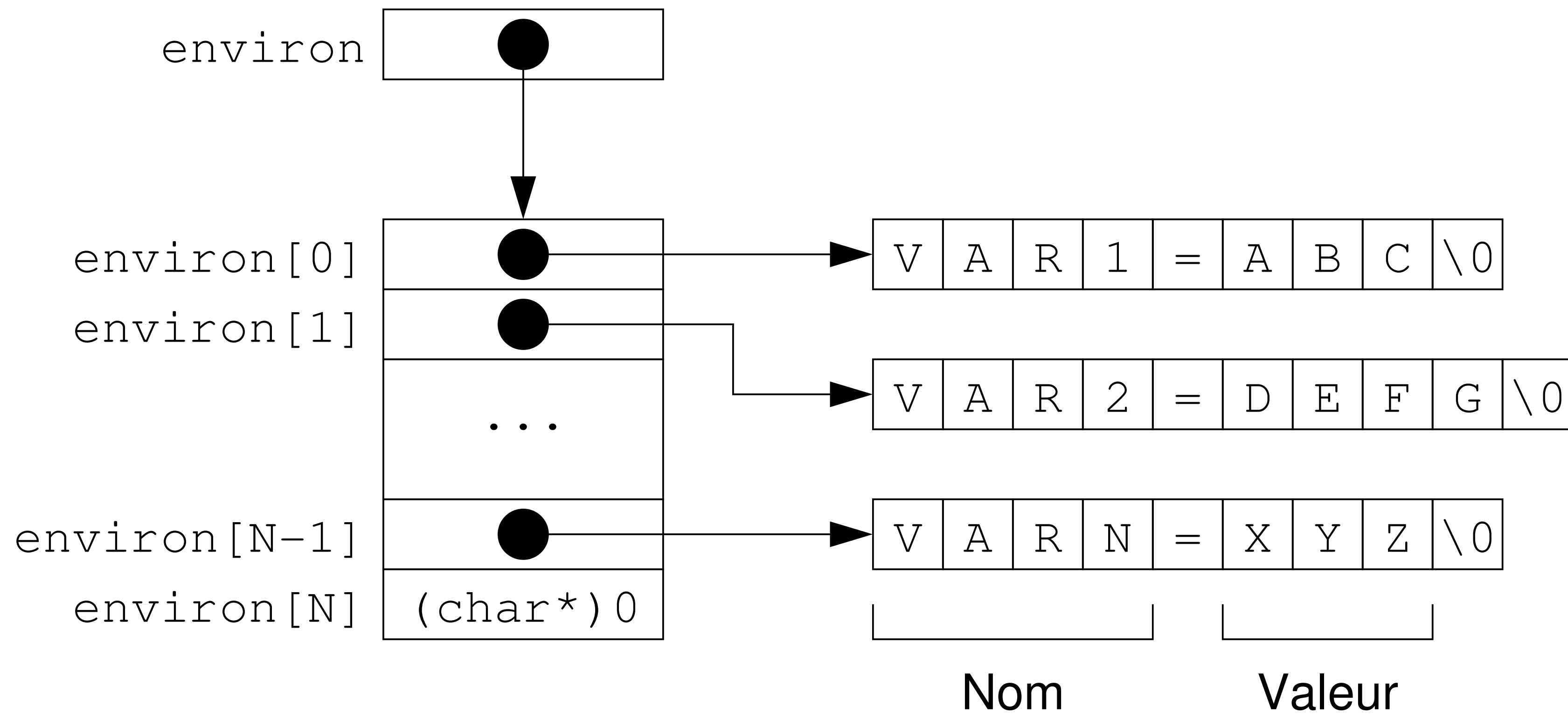
◇ Solution recommandée par *Posix*

◇ `extern char ** environ;`

◇ Initialisée à la création du processus

◇ Même structure que l'argument `envp` de `main()`

Accès à l'environnement



Accès à l'environnement

```c
#include <stdio.h>

extern char ** environ;

int main(void)
{
for(int i=0;environ[i];i++)
  {
  fprintf(stderr,"%d : %s\n",i,environ[i]);
  }
return(0);
}
```

```
0 : PWD=/home/harrouet/tmp
1 : COLORFGBG=default;default
2 : XAUTHORITY=/home/harrouet/.Xauthority
3 : WINDOWID=35651586
4 : LC_MESSAGES=en_US
5 : HOSTNAME=nowin
...
42 : RPM_INSTALL_LANG=en
43 : LC_COLLATE=en_US
44 : _=./a.out
```

Accès à l'environnement

- **Modifier l'environnement avec** `environ`
  - Modification directe des chaînes, du tableau, ou du pointeur
  - Peu aisé !
- **La fonction** `getenv()` (`man 3 getenv`)
  - ```
    #include <stdlib.h>
    char * getenv(const char * name);
    ```
  - Retourne la valeur de la variable `name`
    - Pointeur nul si variable non trouvée
    - Chaîne `""` si la variable est vide
    - Chaîne allouée statiquement par `getenv()`
      - Ne pas modifier son contenu (en faire une copie) !
      - Ne pas libérer sa mémoire !
    ```

Accès à l'environnement

▷ **La fonction** `setenv()` (`man 3 setenv`)

◇
```
#include <stdlib.h>
int setenv(const char * name,const char * value,
           int overwrite);
```

◇ Écriture dans l'environnement d'une variable `name` valant `value`

◇ Si la variable `name` existe déjà

- ∘ Modifier sa valeur si `overwrite` est non nul
- ∘ La laisser inchangée sinon (`value` inutilisée)

◇ Si elle n'existe pas → la créer

◇ Retour : `0` si OK, `-1` si erreur (manque de mémoire)

Accès à l'environnement

▷ **La fonction** `unsetenv()` (`man 3 unsetenv`)

◇
```
#include <stdlib.h>
void unsetenv(const char * name);
```

◇ Retire la variable `name` de l'environnement

▷ **La fonction** `putenv()` (`man 3 putenv`)

◇
```
#include <stdlib.h>
int putenv(const char * string);
```

◇ Ajout d'une chaine `NOM=VALEUR` à l'environnement

◇ Écrase la variable `NOM` si elle existe déjà

◇ Retour : `0` si OK, `-1` si erreur (manque de mémoire)

Accès à l'environnement

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

int main(int argc,char ** argv)
{
for(int i=1;i<argc;i++)
  {
  const char * name=argv[i];
  char * value=getenv(name);
  if(value) fprintf(stderr,"%s=%s\n",name,value);
  else fprintf(stderr,"Unknown variable %s !\n",name);
  }
return(0);
}

$ ./prog DISPLAY LOGNAME TOTO HOSTNAME
DISPLAY=:0.0
LOGNAME=harrouet
Unknown variable TOTO !
HOSTNAME=nowin
```

Les classes de variables du *C*

▷ **Les variables globales**

- ◇ Accessibles depuis l'ensemble du programme
- ◇ Existent tout au long de l'exécution
- ◇ Initialisées (explicitement ou à `0`) au démarrage

▷ **Les variables statiques**

- ◇ Visibles uniquement dans leur bloc de code
- ◇ Existent tout au long de l'exécution
- ◇ Initialisées (explicitement ou à `0`) au démarrage
- ◇ Différentes des variables globales déclarées `static` (visibilité inter-fichiers)

Les classes de variables du *C*

- **Les variables locales (automatiques)**
  - Visibles uniquement dans leur bloc de code
  - N'existent que de l'entrée à la sortie du bloc
  - Pas d'initialisation implicite
- **Les paramètres**
  - Visibles uniquement dans la fonction ($\simeq$ variables locales)
  - N'existent que de l'entrée à la sortie de la fonction
  - Initialisés par recopie des valeurs externes
- **Les variables temporaires**
  - Pas directement manipulables
  - Résultat intermédiaire d'une opération complexe
  - Durée de vie éphémère

Les classes de variables du *C*

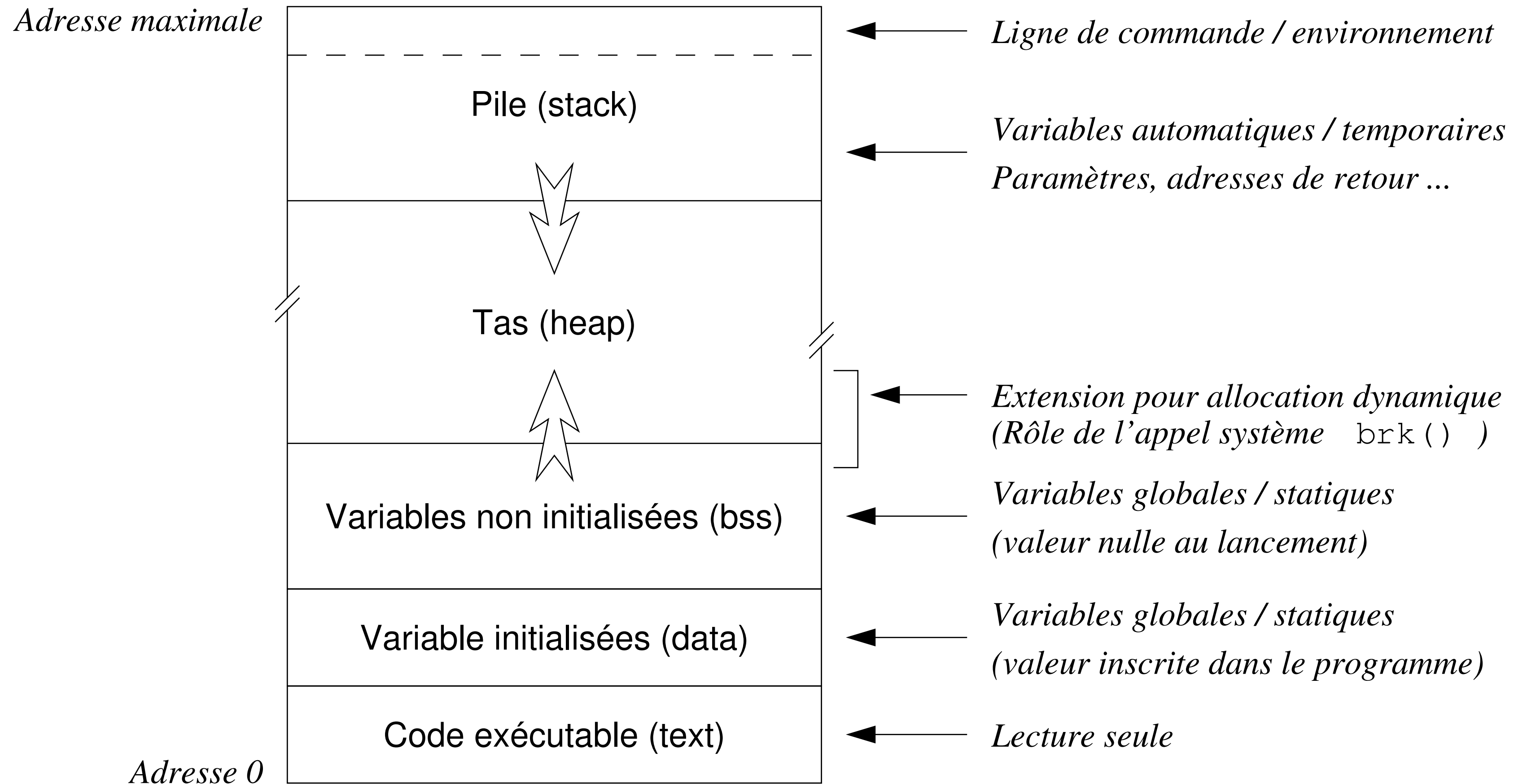
```
int var1=1;                   // variable globale explicitement initialisee
int var2;                     // variable globale implicitement initialisee a 0

static int var4;              // variable globale (visibilite limitee au fichier)
extern int var5;              // variable globale (definie dans un autre fichier)

void f(int arg1,int * arg2) // parametres
{
static int var6=6;            // variable statique
int var7=arg1*2+5;            // variable locale & variable temporaire
arg1++;                       // modification locale du parametre
(*arg2)++;                    // modification de la variable designee par le parametre
}

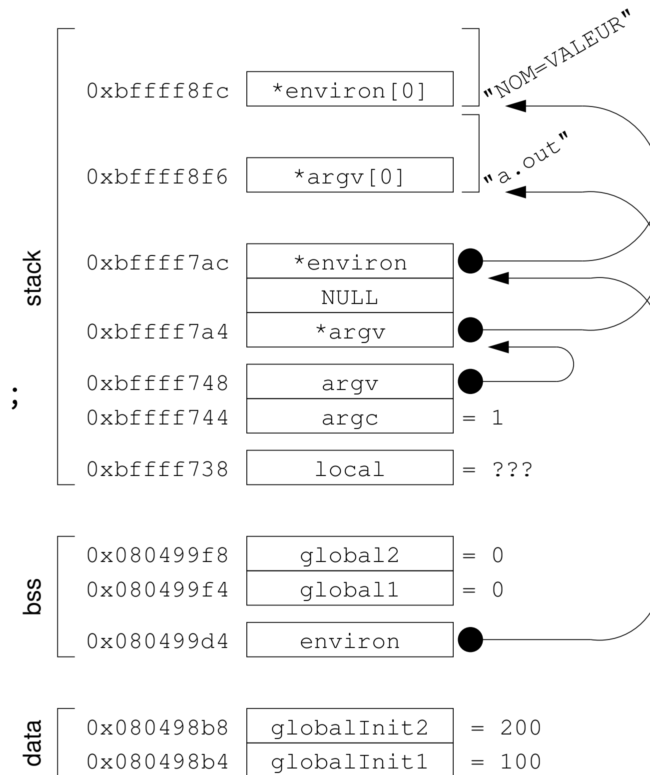
int main(void)
{
int var8=8;                   // variable locale
f(12*8,&var8);                // variables temporaire
return(0);
}
```

Espace mémoire d'un processus



Espace mémoire d'un processus

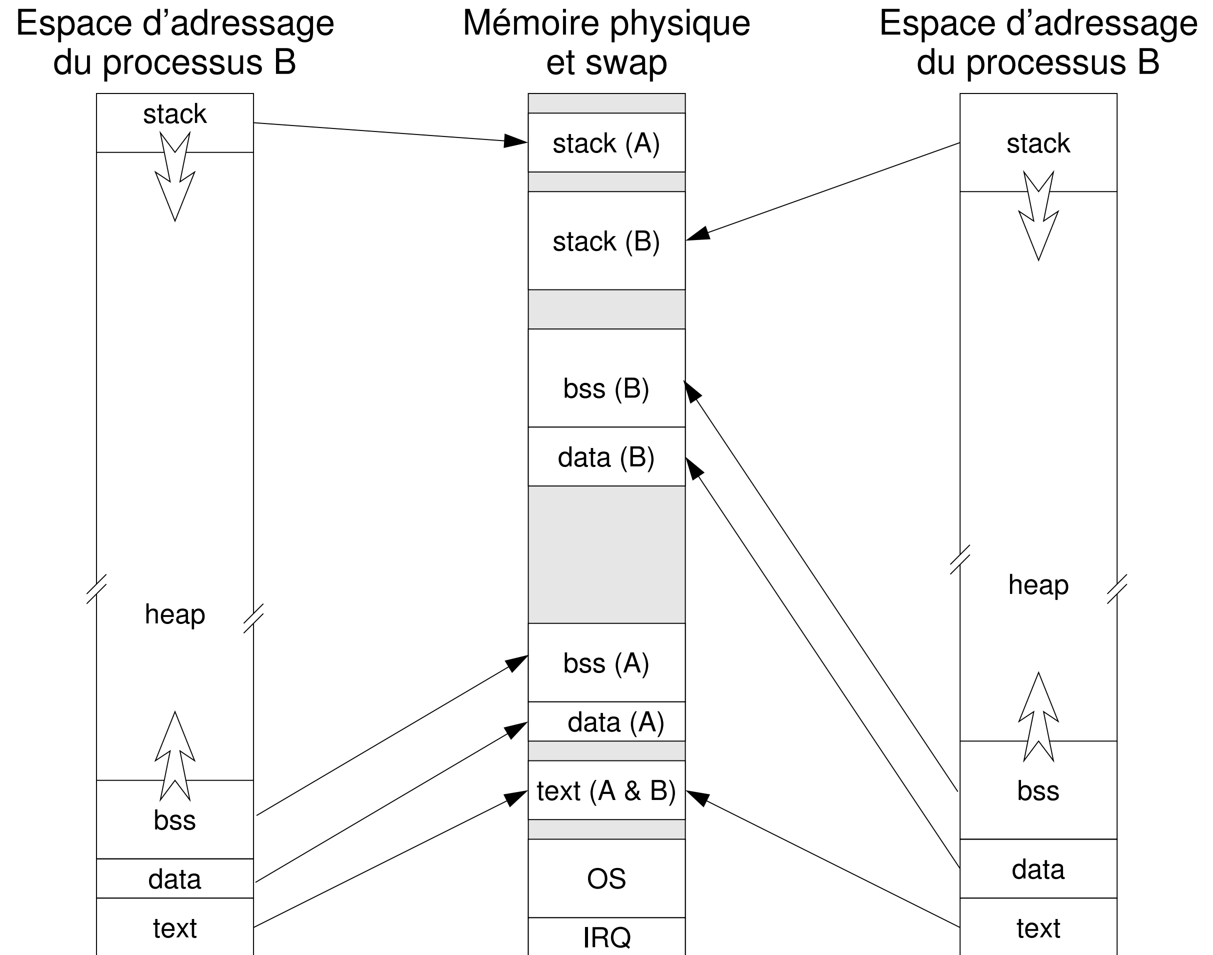
```
#include <stdio.h>
#include <stdlib.h>
int global1, globalInit1=100;
int global2, globalInit2=200;
extern char ** environ;
int main(int argc,char ** argv)
{
int local;
fprintf(stderr,"*environ[0] at 0x%.8lx\n",environ[0]);
fprintf(stderr,"   *argv[0] at 0x%.8lx\n",argv[0]);
fprintf(stderr,"   *environ at 0x%.8lx\n",environ);
fprintf(stderr,"      *argv at 0x%.8lx\n",argv);
fprintf(stderr,"       argv at 0x%.8lx\n",&argv);
fprintf(stderr,"       argc at 0x%.8lx\n",&argc);
fprintf(stderr,"      local at 0x%.8lx\n",&local);
fprintf(stderr,"    global2 at 0x%.8lx\n",&global2);
fprintf(stderr,"    global1 at 0x%.8lx\n",&global1);
fprintf(stderr,"    environ at 0x%.8lx\n",&environ);
fprintf(stderr,"globalInit2 at 0x%.8lx\n",&globalInit2);
fprintf(stderr,"globalInit1 at 0x%.8lx\n",&globalInit1);
return(0);
}
```



Espace mémoire d'un processus

Ici, les processus A et B

exécutent le même programme



Allocation dynamique de mémoire

- **L'appel système** `brk()` (`man 2 brk`)
  - Déplacer la limite du segment `DATA/BSS` vers le tas
  -
    ```
    #include <unistd.h>
    int brk(void * segmentEnd);
    void * sbrk(ptrdiff_t increment); // fonction
    ```
  - Seule chose connue du système : taille des segments
  - Le processus peut faire ce qu'il veut de cet espace
    - Pas de structure *a priori*
    - N'importe quelle opération est autorisée à l'intérieur
  - Éviter d'utiliser directement cet appel !

Allocation dynamique de mémoire

▷ **Les fonctions d'allocation** (`man 3 malloc`)

- Utilisent les appels `brk()` et `mmap()` en interne
- Gèrent elles même l'occupation de l'espace obtenu
-
```
#include <stdlib.h>
void * malloc(size_t size);
void * calloc(size_t nb,size_t size);
void * realloc(void * ptr,size_t size);
void free(void * ptr);
```
- `malloc()` : allouer un bloc de `size` octets
- `calloc()` : allouer un bloc de `nb*size` octets (initialisés à `0`)
- `realloc()` : redimensionner un bloc mémoire (conserve le contenu)
- `free()` : libérer un bloc **obtenu par ces fonctions**

Allocation dynamique de mémoire

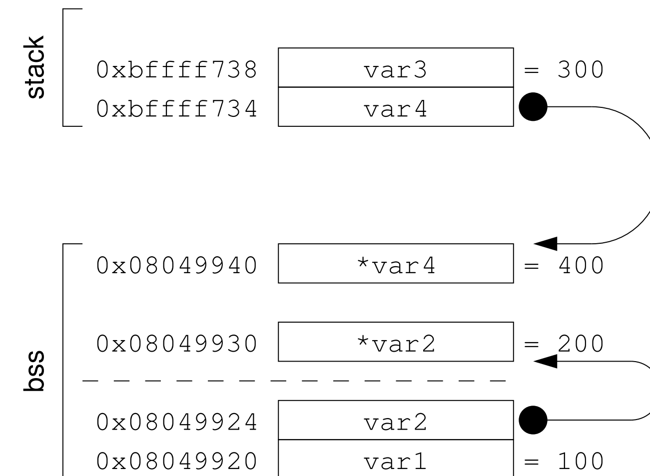
▷ **Les fonctions d'allocation**

- ◇ Retour d'un pointeur nul si allocation impossible (rare !)
- ◇ Remarques sur `realloc()` :
  - ◦ `realloc(ptr,0);` ≡ `free(ptr);`
    `ptr=realloc((void *)0,size);` ≡ `ptr=malloc(size);`
  - ◦ Le bloc peut être déplacé → mise à jour des pointeurs !
- ◇ Ces fonctions sont configurables
- ◇ Outils intervenant à l'intérieur de ces fonctions :
  - ◦ Tenir des statistiques sur leur utilisation (*mtrace* . . .)
  - ◦ Mettre en évidence des incohérences (*mcheck*, *Electric Fence* . . .)
- ◇ Différent de `new`/`delete` de *C++*
  - ◦ Traitement supplémentaires pour initialisation/destruction

Allocation dynamique de mémoire

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

int var1, * var2;
int main(void)
{
int var3, * var4;
var1=100;
var2=(int *)malloc(sizeof(int));
*var2=200;
var3=300;
var4=(int *)malloc(sizeof(int));
*var4=400;
fprintf(stderr," var1 at 0x%.8lx (%d)\n",&var1,var1);
fprintf(stderr," var2 at 0x%.8lx\n",&var2);
fprintf(stderr,"*var2 at 0x%.8lx (%d)\n",var2,*var2);
fprintf(stderr," var3 at 0x%.8lx (%d)\n",&var3,var3);
fprintf(stderr," var4 at 0x%.8lx\n",&var4);
fprintf(stderr,"*var4 at 0x%.8lx (%d)\n",var4,*var4);
free(var2); free(var4);
return(0);
}
```



Allocation dynamique de mémoire

▷ **Allocation dynamique dans la pile**

- ◇ Travailler très localement sur un volume variable de données
- ◇ Plus rapide que `malloc()`
- ◇ La fonction `alloca` (`man 3 alloca`)

```
#include <stdlib.h>
#include <alloca.h>
void * alloca(size_t size);
```

- ◇ Bloc valide jusqu'à la sortie de la fonction appelante
- ◇ Destruction automatique du bloc à la sortie
  - ◦ Ne pas utiliser `free()` ou `realloc()`
  - ◦ Ne pas référencer ce bloc à l'extérieur