La Gestion des Processus

Dans les Systèmes UNIX

Michel Soto

Université Paris Descartes

Processus

Definition

Processus: programme en cours d'exécution

REMARQUE

- **Plusieurs instances de processus** peuvent exister **pour un même programme**

Contexte d'un processus

- **Mémoire** : 3 segments (code, pile d'exécution et tas, données),
- **Environnement** : variables d'environnement (variable = valeur),
- **Identifiants** : (UID, GID),
- **État des fichiers ouverts**,
- **Contexte matériel** : mot d'état (PSW), CP, SP, registres, ...

Type de processus

Definition

- **Processus système**
  Attaché à aucun terminal, ils est créé par :
  - le noyau : scheduler, pagedaemon, ...
  - init (`/etc/int`) : démons `lpd`, `ftpd`, ...
- **Processus utilisateur**
  Lancé par un utilisateur depuis un terminal.

Mode d'exécution

Definition

- **Mode utilisateur**
  Le processus exécute ses instructions et utilise ses propres données.
- **Mode noyau (ou système)**
  Le processus exécute les instructions du noyau.

La fonction `main ()`

Un processus débute par l'exécution de la fonction `main()` du programme correspondant

Definition

```
int main (int argc, char *argv[]);
ou
int main (int argc, char **argv);
```

- `argc`: nombre d'arguments de la ligne de commande y compris le nom du programme
- `argv[]`: tableau de pointeurs vers les arguments (paramètres) passés au programme lors de son lancement

A NOTER

- `argv[0]` pointe vers le nom du programme
- `argv[argc]`vaut `NULL`

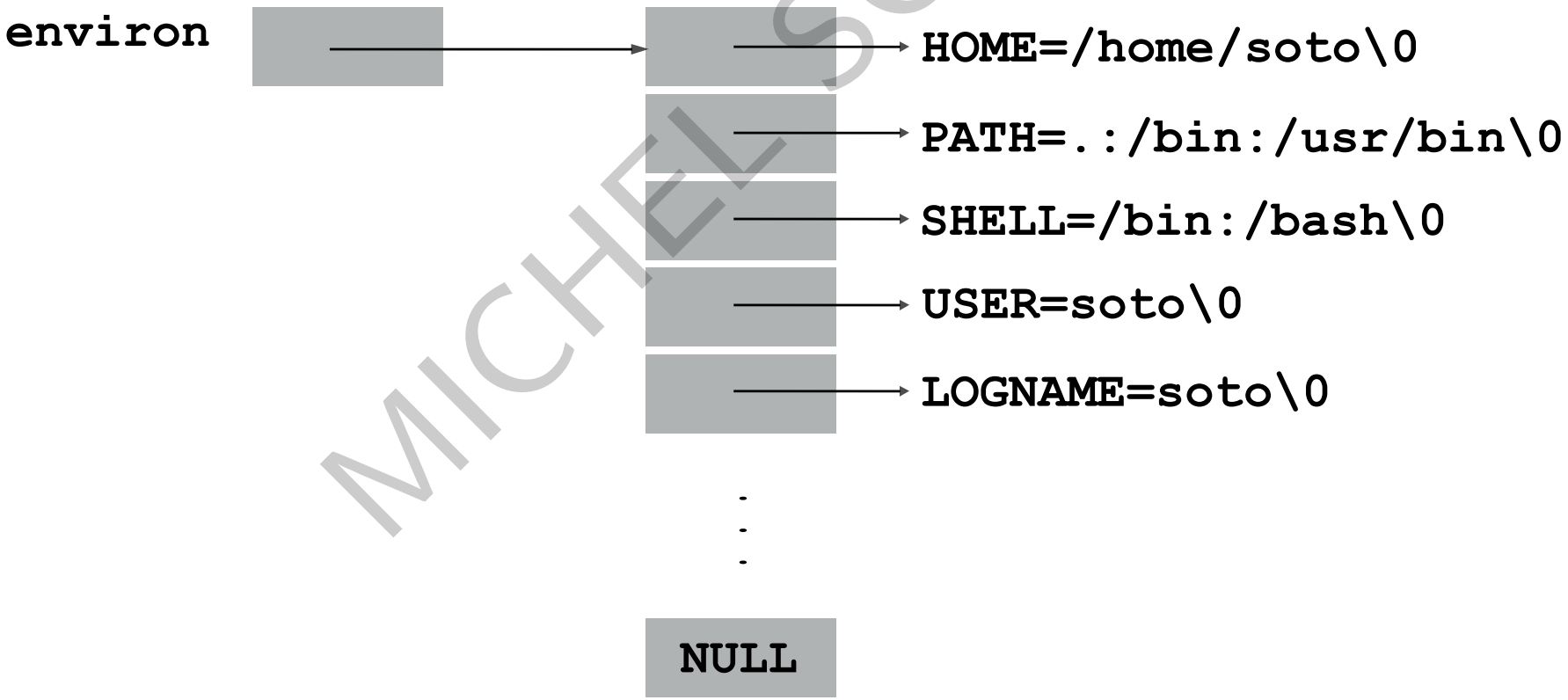
Utilisation des arguments de la fonction `main ()`

Un exemple

```
#include <stdio.h>

int main (int argc, char *argv[]){
int i;
for (i=0;i<argc;i++){
    printf ("argv[%d]: %s\n",i, argv[i]);
}
return (0);
}// main

>./affich_param Bonjour à tous
argv[0]: ./affich_param
argv[1]: Bonjour
argv[2]: à
argv[3]: tous
```

Variables d'environnement de la fonction `main ()`

- Une liste de variables d'environnement est passée au programme lors de son lancement sous forme d'un tableau de pointeurs de caractères.
- Chaque variable est de la forme `nom=valeur`.
- Une variable globale `environ` contient l'adresse de ce tableau. `extern char **environ`



Représentation en mémoire d'un processus

- Segment de texte (`text`).
  - Contient les instructions en langage machine du programme
  - Partageable: un seul exemplaire par programme est nécessaire en mémoire
  - Accessible uniquement en lecture afin d'éviter toute modification accidentelle
- Segment des variables initialisées (`data segment`)
  Variables en dehors de toute fonction qui ont été initialisées au moment de leur déclaration

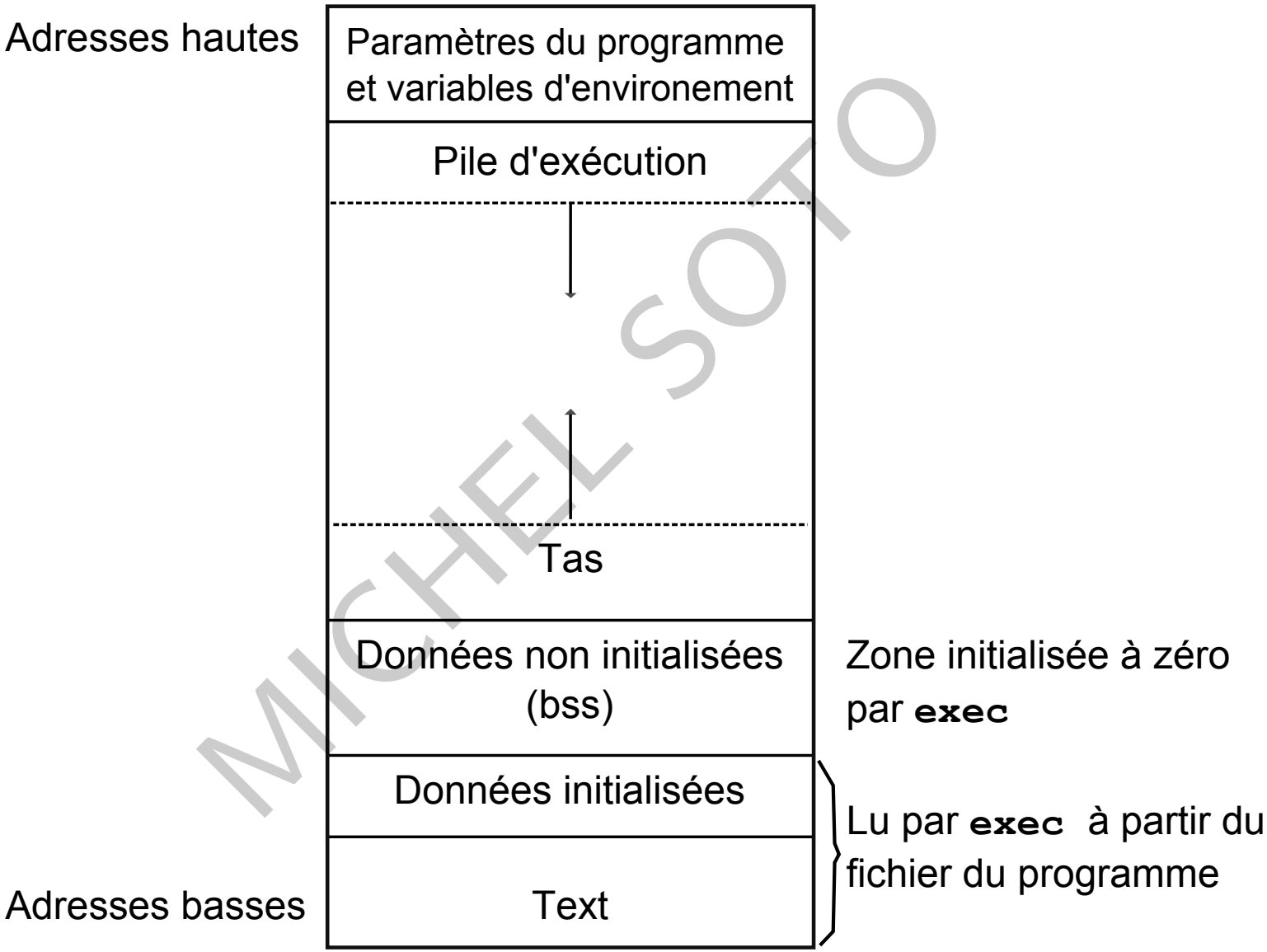
Exemple

```
#include <stdio.h>

int angle=45; // Dans le data segment du processus

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

Représentation en mémoire d'un processus (Suite)

- Segment des variables non initialisées (`uninitialized data segment`)
  - Les variables déclarées en dehors de toute fonction sont initialisées avec un zéro arithmétique ou à `NULL` pour les pointeurs
  - Appelé historiquement `block started by symbol (bss)`
- Pile d'éxecution (`stack`).
  - Variables automatiques temporaires créées à l'appel de chaque fonction
  - Adresse de retour de la fonction
- Tas (`heap`)
  Utilisé pour l'allocation dynamique de mémoire (`malloc, calloc, realloc, free`)

Représentation en mémoire d'un processus (Suite)



Représentation en mémoire d'un processus (Fin)

La commande size

```
> size /bin/bash /bin/ps
   text    data     bss     dec     hex filename
 867835   35864   22880  926579   e2373 /bin/bash
  80115    1472  132184  213771   3430b /bin/ps
```

Les bibliothèques partagées

- Des bibliothèques de fonctions sont utilisées par la grande majorité des processus
  - Elles sont donc dupliquées dans chaque exécutable
- Les bibliothèques partagées permettent d'éviter cette duplication.
  - Une seule copie de chaque bibliothèque est présente en mémoire
    - + Le code de chaque exécutable est réduit
    - + Les nouvelles versions de bibliothèques ne nécessitent pas la réédition de liens des programmes qui les utilisent tant que la signature des fonctions ne change pas
    - − Le temps d'exécution est allongé au premier appel du programme et lors du premier appel de chaque fonction de la bibliothèque

Exemple

```
#include <stdio.h>
main(){ printf ("bonjour\n"); }
> cc bonjour.c
> size a.out
   text    data     bss     dec     hex filename
    987     252       8    1247     4df a.out
> cc -static bonjour.c
> size a.out
   text    data     bss     dec     hex filename
 577600    1928    7016  586544   8f330 a.out
```

Allocation de mémoire

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

Alloue `size` octets de mémoire
Le contenu initial de la mémoire allouée est indéterminé

- `size`: taille en octet de la zone de mémoire désirée

Retourne: un pointeur non NULL en cas de succès ou un pointeur NULL en cas d'échec

```
#include <stdlib.h>
void *calloc(size_t nobj, size_t size);
```

Alloue `nobj` objets de `size` octets de mémoire chacun
Toute la zone de mémoire allouée est initialisée à 0

- `nobj`: nombre d'objets
- `size`: taille en octet de l'objet

Retourne: un pointeur non NULL en cas de succès ou un pointeur NULL en cas d'échec

Allocation de mémoire (Fin)

```
#include <stdlib.h>
void *realloc(void *ptr, size_t newsize);
```

Augmente ou diminue la taille d'une zone de mémoire précédemment allouée

- `ptr`: pointeur vers la zone de mémoire à augmenter ou à diminuer
- `newsize`: nouvelle taille en octet de la zone mémoire

Retourne: un pointeur non NULL en cas de succès ou un pointeur NULL en cas d'échec

Le contenu de la mémoire reste inchangé sur le minimum entre l'ancienne et la nouvelle taille de la zone allouée. En cas d'augmentation, le supplément de mémoire alloué n'est pas initialisé

```
#include <stdlib.h>
void free(void *ptr);
```

Libère la zone de mémoire pointée par `ptr`

- `ptr`: pointeur vers la zone de mémoire à libérer
- `newsize`: nouvelle taille en octet de la zone mémoire

Le contenu de `ptr` doit être le résultat d'un appel précédent à `malloc`, `calloc` ou `realloc`

Identification des processus

Definition

Identificateur de processus (`Process ID` ou `PID`) :
Entier *unique supérieur à zéro* attribué par le système à tout processus

Propriété

Lorsqu'un processus se termine son `PID` redevient utilisable pour un nouveau processus après *un délai de garde*

Qui suis-je ?

Un processus ne peut connaître son `PID` qu'en s'adressant au système avec la primitive `getpid`

- `#include <unistd.h>`
  `pid_t getpid(void);`
  Renvoie le `PID` du processus appelant

Ne retourne pas d'erreur

Processus particuliers

- Processus 0 : `scheduler` ou `swapper`
  Aucun programme sur le disque ne correspond à ce processus. Il s'agit d'un processus système qui fait partie du noyau
- Processus 1 : `init`
  - Crée par le système à la fin du démarrage (`bootstrap`)
    Le programme correspondant est `/etc/init` sur les anciennes versions d'Unix et sur les récentes `/sbin/init`
  - Il utilise les fichiers d'initialisation du système `/etc/rc*` ou `/etc/init.d` et `/etc/inittab`
  - Il n'appartient pas au noyau.
  - Il ne meurt jamais et devient le père de tout processus orphelin

Autres identifiants

```
#include <unistd.h>
```

- `pid_t getppid(void);`
  Renvoie le PID du père du processus appelant: *qui est mon père ?*
- `uid_t getuid(void);`
  Renvoie le groupe réel (RGUID) du processus appelant
- `uid_t geteuid(void);`
  Renvoie le groupe effectif (EGUID) du processus appelant
- `uid_t getgid(void);`
  Renvoie le groupe réel (RGUID) du processus appelant
- `uid_t getegid(void);`
  Renvoie le groupe effectif (EGUID) du processus appelant

Remarque

Aucune de ces primitives ne retourne d'erreur

Création d'un processus

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

Crée un nouveau processus
Retourne :

- Dans le processus créé et appelé *processus fils*: **ZÉRO**
- Dans le processus appelant et appelé *processus père*: le `PID` **du fils créé**
- En cas d'erreur, chez le processus père: -1

Particularités

- UN APPEL RÉUSSI de cette primitive RETOURNE DEUX FOIS:
  une fois chez le père et une fois chez le fils
- UN APPEL RÉUSSI retourne DEUX RÉSULTATS DIFFÉRENTS
- Un père peut avoir plusieurs fils et un fils n'a qu'un seul père
- Un fils a toujours père (adoption par `init`)

Création d'un processus (Suite)

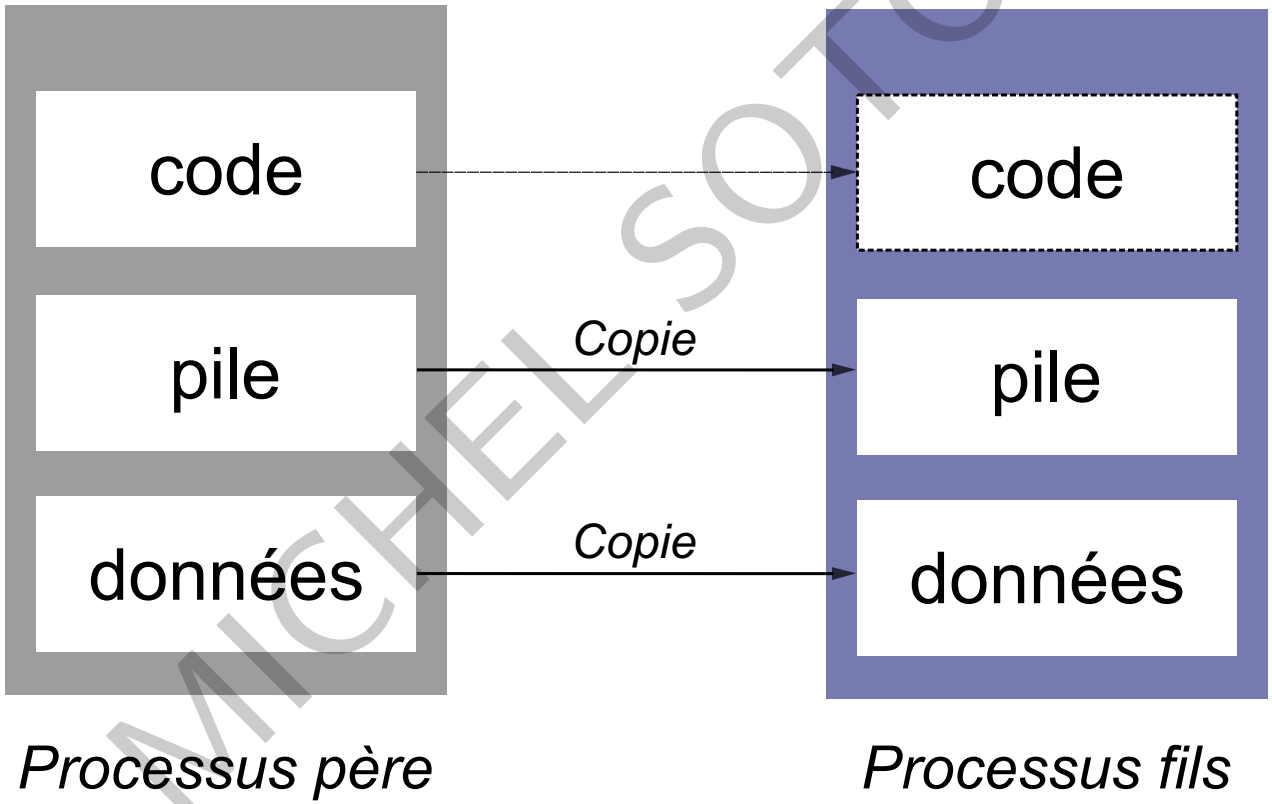
Effets conceptuels du `fork`

- Le fils est une **copie intégrale du père** (clone) **à l'instant du `fork`**
- Le père et le fils continuent leurs exécutions respectives avec l'instruction qui suit immédiatement le `fork` dans le code
- Le père est le fils **ne partagent aucune zone de mémoire** excepté le segment *text*
  - Deux pointeurs ayant la même valeur chez le père et chez le fils désignent deux zones de mémoire physique différentes

Ressource pédagogique complémentaire

- https://youtu.be/n01cCfnMHvk

Création d'un processus (Suite)



Effet du fork

Création d'un processus (Suite)

Exemple

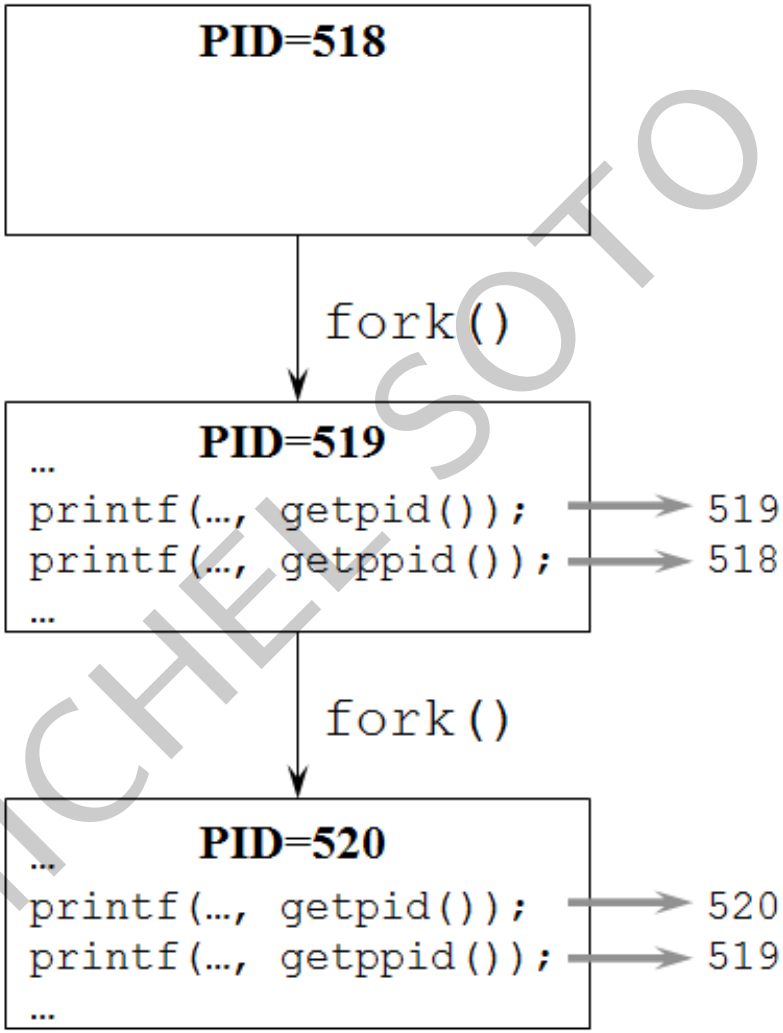
```
#include <unistd.h>
#include <stdio.h>
void main() { fork ();
              printf("Bonjour\n");
}
>a.out
Bonjour
Bonjour
```

```
#include <unistd.h>
#include <stdio.h>
void main() {
pid_t pid;
 pid=fork ();
 if (pid == 0) printf("Bonjour, je suis le fils: %d\n", getpid());
 else printf("Bonjour, je suis le père: %d\n", getpid());
 printf("FIN: %d\n", getpid());
}
```

```
 >a.out
Bonjour, je suis le fils: 6232
Bonjour, je suis le père: 6231
FIN: 6231
FIN: 6232
```

```
 >a.out
Bonjour, je suis le père: 6245
FIN: 6245
Bonjour, je suis le fils: 6246
FIN: 6246
```

Création d'un processus (Suite)



Filiation des processus

Création d'un processus (Suite)

Race condition

Dans une même application, il y a une *race condition* lorsque un résultat diffère selon l'ordre d'exécution des processus qui la composent

- Il est impossible de prévoir qui du père ou du fils s'exécutera en premier après le `fork`
  - Le système décide qui du père ou du fils reprendra en premier son exécution en fonction de sa charge courante et de sa politique globale d'ordonnancement
- Le système fournit des outils de communication inter-processus (signaux, sémaphores, tubes, etc) afin d'éviter les situations de race condition

Création d'un processus (Suite)

Implémentation du `fork`

Un `fork` est très souvent suivi d'un `exec` (écrasement du code) rendant toute copie inutile et pénalisante

- copy-on-write (COW)
  - En réalité, le père et le fils partagent l'espace mémoire du père en lecture seule
  - La copie n'est effectuée que lorsque le père ou le fils tentent de modifier cet espace partagé et seule la page concernée est copiée
- `vfork`
  - Aucune copie n'est effectuée
  - Le père et le fils partagent **intégralement** l'espace mémoire du père en lecture **ET** écriture

Création d'un processus (Suite)

Attributs hérités par le fils

- Descripteurs des fichiers ouverts,
- `UID réel`, `GID réel`, `UID effectif`, `GID effectif`,
- Répertoire de travail courant,
- Flags `set-user-ID` et `set-group-ID`,
- Masque de création des fichiers,
- Masques des signaux,
- Flag `close-on-exec` pour tout descripteur de fichier ouvert,
- Variables d'environnement,
- Segments de mémoire attachés, ...

Création d'un processus (Fin)

Attributs non hérités par le fils

- Valeur de retour du `fork`
- Identifiant (`PID`) du processus père
- Verrous posés par le processus père
- Ensemble des signaux pendants initialisé à vide
- Compteurs `tms_utime`, `tms_stime`, `tms_cutime` remis à 0

Mise en sommeil d'un processus

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

- Suspend le processus jusqu'à ce que :
  - la totalité du temps spécifié par `seconds` se soit écoulé
    ou
  - un signal non ignoré soit reçu par le processus

Retourne : 0 ou le nombre de secondes restant avant la fin du délai

ATTENTION

Dans certaines implémentations, `sleep()` est implémentée avec la primitive `alarm()`(SVR4) (voir signaux). Il y a des possibilités d'interférences en cas d'utilisation conjointe de ces deux fonctions.

Mise en sommeil d'un processus (Suite)

```
#include <time.h>
int nanosleep(const struct timespec *req, struct timespec *rem);

struct timespec {
       time_t tv_sec;        /* secondes */
       long   tv_nsec;       /* nanosecondes */
       };
```

Suspend le processus jusqu'à ce que la totalité du temps spécifié par `req` se soit écoulé ou qu'un signal non ignoré soit reçu par le processus

- `*req` : délai
- `*rem` : temps restant avant la fin du délai (`NULl` possible)

Retourne : 0 si l'appel n'a pas été interrompu

Cette fonction ne repose pas sur les signaux et peut être utilisée sans aucun souci d'interaction indésirable avec d'autres fonctions.

Mise en sommeil d'un processus (Fin)

Exemple

```
void main() {
int pid,fd;

fd = open("fich", O_RDWR|O_CREAT, 0666);
if ((pid = fork()) == -1) {
        perror("Erreur au fork");
        exit(1);
}

if ( pid == 0 ) {
   printf("processus fils, mon pid=%d, pid de mon père=%d",getpid(),getppid());
   write(fd,"12345", 5);
   exit(0);
}

printf("processus pere, pid du fils = %d", pid);
sleep(10);
write(fd, "6789", 4);
...
}
```

Contenu de `fich` : "123456789"

Terminaison d'un processus

```
#include <stdlib.h>
void exit(int status);
```

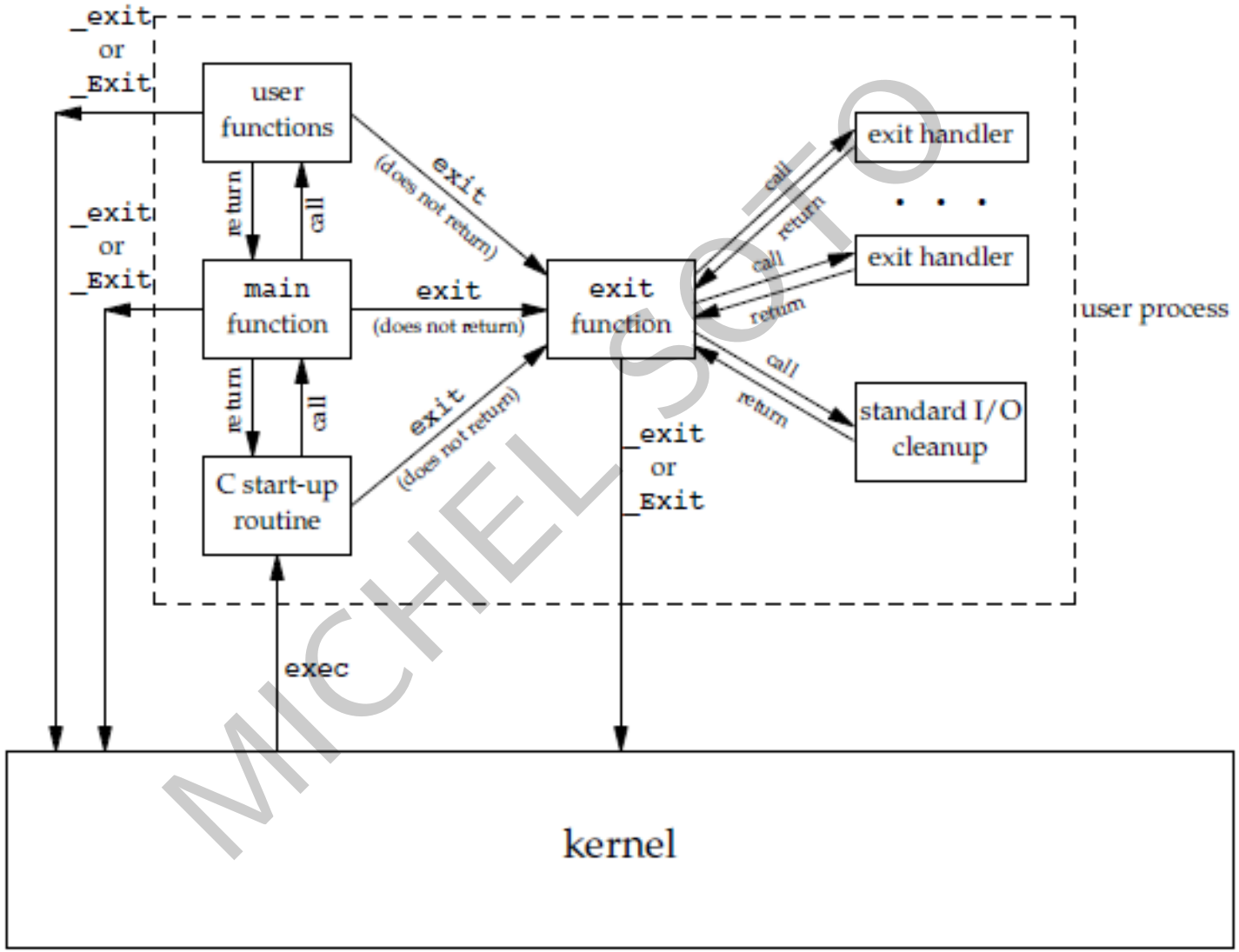
```
#include <unistd.h>
void _exit(int status);
```

1. Ferment tous les descripteurs de fichiers
2. Terminent le processus appelant
3. Transmet au père la valeur de `status`

Différences

- `exit` exécute les gestionnaires de fin installés avec `atexit` alors que `_exit` n'exécute aucun gestionnaire de fin ou de signal. `exit` appelle `_exit`
- `exit` appartient à la librairie standard du C alors que `_exit` est une primitive du système
- La vidange ou pas des E/S standard dépend des implémentations

Terminaison d'un processus



from *Advanced Programming in the UNIX® Environment*

Priorité des processus

```
#include <sys/time.h>
#include <sys/resource.h>
int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

Connaitre ou modifier la priorité d'un processus ou d'un ensemble de processus

`who` est interprété relativement à `which` :

`which`	`who`
`PRIO_PROCESS`	`process ID`
`PRIO_PGRP`	`group ID`
`PRIO_USER`	`user ID`

- Si `who` $= 0$: processus ou groupe ou utilisateur courant (respectivement à `which`
- Les valeurs de `prio` $\in [-20(max), +19(min)]$.
- Seul le super utilisateur peut diminuer la valeur numérique de la priorité et donc augmenter la priorité d'un processus.

Priorité des processus (Suite)

```
#include <sys/time.h>
#include <sys/resource.h>
int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

`getpriority()` :
Retourne :

- la plus haute priorité du ou des processus spécifiés par `which`
- -1 en cas d'erreur **ou de priorité égale à -1**.
  Pour faire la distinction il faut mettre `errno` à zéro avant l'appel
  Après l'appel, si `errno` $\neq 0$ il s'agit d'une erreur sinon il s'agit de la priorité

`setpriority()` : modifie la priorité du ou des processus spécifiés par `which` Retourne :

- 0 en cas de succès
- -1 en cas d'erreur.

Priorité d'un processus (Fin)

```
#include <unistd.h>
int nice(int inc);
```

Modifie la priorité ("politesse") du processus appelant

- `inc` : valeur ajoutée à la priorité courante
  Le système ajuste le résultat aux valeurs autorisées pour le processus

Retourne :

- La nouvelle valeur de priorité du processus appelant
- -1 en cas d'erreur **ou de priorité égale à -1**.
  Pour faire la distinction il faut mettre `errno` à zéro avant l'appel
  Après l'appel, si `errno` $\neq 0$ il s'agit d'une erreur sinon il s'agit de la priorité

Temps CPU d'un processus

```
#include <sys/times.h>
clock_t times(struct tms *buf);

struct tms {
      clock_t   tms_utime;  /* temps CPU en mode utilisateur */
      clock_t   tms_stime;  /* temps CPU en mode système */
      clock_t   tms_cutime; /* temps CPU utilisateur des fils terminés et attendus */
      clock_t   tms_cstime; /* temps CPU système des fils terminés et attendus */
};
```

Fournit le temps CPU consommé dans les deux modes (utilisateur et noyau) par le processus et ses fils terminés.
Retourne :

- le temps global écoulé exprimé en nombre de tics d'horloge
- -1 en cas d'erreur

REMARQUE

Pour convertir les résultats en durées exprimées en secondes, les diviser par `CLK_TCK` (`<limits.h>`)

Synchronisation père-fils

- L'exécution d'un processus père et de ses processus fils sont totalement asynchrones
- La fin d'un fils peut se produire à n'importe quel moment de l'exécution de père
- Le système informe le père de la fin d'un de ses fils en lui envoyant le signal `SIGCHLD`
  - *Par défaut*, ce signal *n'a aucun effet sur le processus père*
    mais le père peut décider de capter ce signal
- Un processus père peut attendre et s'informer de la terminaison de ses fils grâce aux fonctions `wait` et `waitpid`
  - Ces primitives fonctionnent en tandem avec les fonctions `exit` et `_exit`

Synchronisation père-fils (Suite)

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

- Bloque le processus père si *tous* ses fils sont en cours d'exécution
- Retourne immédiatement si au moins un des fils est terminé ou si tous les fils sont terminés
- `status` : contient la valeur du `exit` d'un fils ou une valeur élaborée par le système en cas de terminaison brutale du fils concerné
- Renvoie :
  - Le `PID` du fils qui s'est terminé
  - -1 en cas d'erreur

Synchronisation père-fils (Suite)

Exemple

```
#include <unistd.h>
#include <stdlib.h> // pour le EXIT_SUCCESS
#include <stdio.h>
void main() {
pid_t pid; int status;
 pid=fork ();
 if (pid == 0) {printf("Bonjour, je suis le fils: %d\n", getpid());
                sleep (5);
                exit (EXIT_SUCCESS);}

// Cette partie du code n'est jamais exécutée par le fils grâce au exit
   printf("Bonjour, je suis le père: %d\n", getpid());
   printf("J'attends la fin de mon fils: %d\n", pid);
   pid=wait (&status);
   printf("Mon fils %d s'est terminé avec la valeur %d\n", pid, status);
}
```

```
 >a.out
Bonjour, je suis le père: 9625
J'attends la fin de mon fils: 9626
Bonjour, je suis le fils: 9626
Mon fils 9626 s'est terminé avec la valeur 0
```

```
 >a.out
Bonjour, je suis le fils: 9629
Bonjour, je suis le père: 9628
J'attends la fin de mon fils: 9629
Mon fils 9629 s'est terminé avec la valeur 0
```

Synchronisation père-fils (Suite)

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

- `pid`
  - $<$ -1 attente de tout processus fils dans le groupe |`pid`|
  - $=$ -1 attente de tout processus fils
  - $=$ 0 attente de tout processus fils du même groupe que l'appelant
  - $>$ 0 attente du processus fils d'identité `pid`
- `options`
  - `WNOHANG` retour immédiat si aucun fils n'est terminé.
  - `WUNTRACED` retour immédiat si un fils est stoppé
  - `WCONTINUED` retour immédiat si un fils stoppé a été relancé par le signal `SIGCONT`

REMARQUE

```
wait(int *status) = waitpid(-1, &status, 0);
```

Interprétation de valeur renvoyée par `exit` dans le paramètre `status`

- La valeur retournée au père par un fils se trouve dans le 2e octet de poids faible du paramètre `status` de `wait` et `waitpid`

Octet 3	Octet 2	Octet 1	Octet 0
			Paramètre du exit

Entier du processus fils

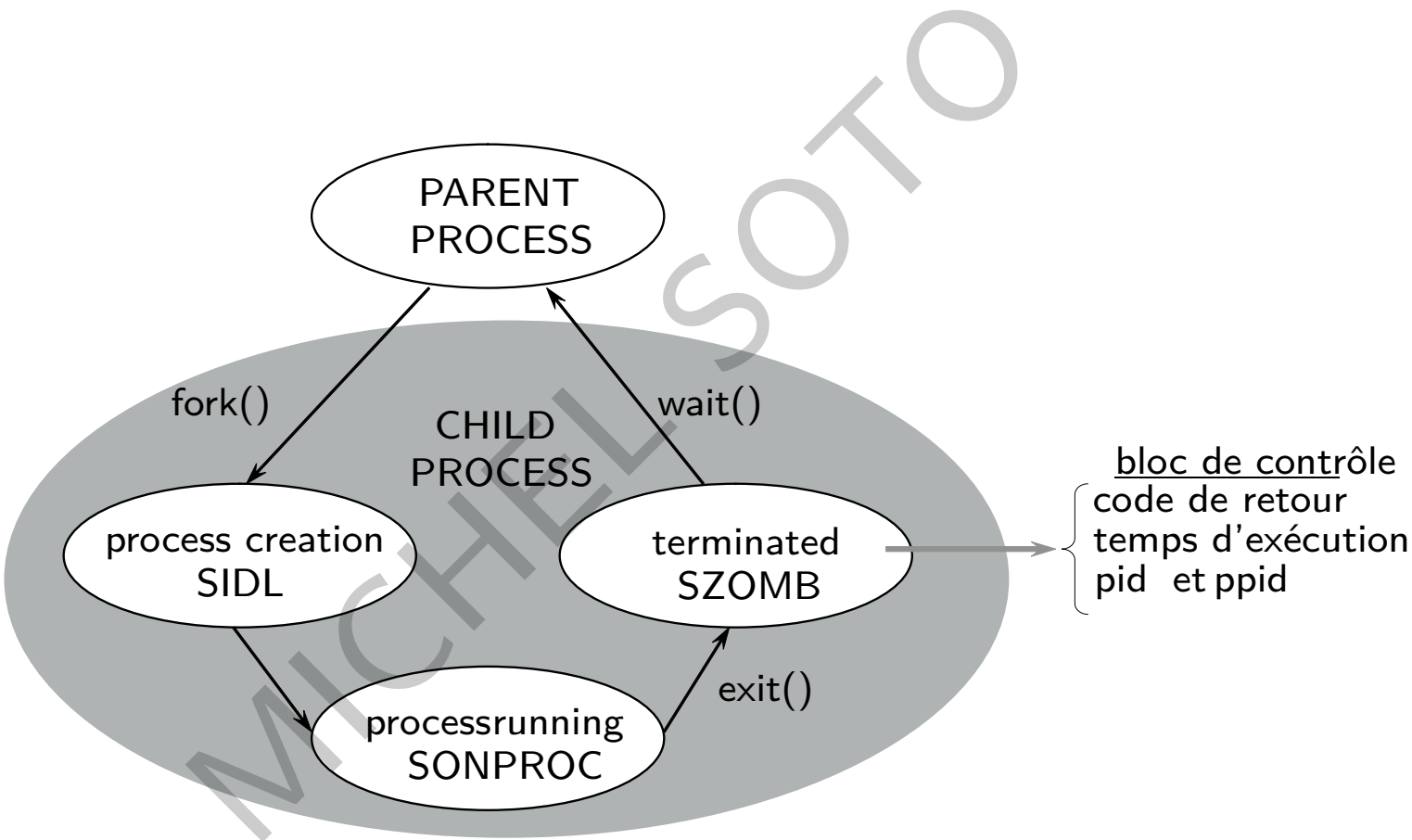
Octet 3	Octet 2	Octet 1	Octet 0
		Paramètre du exit	0

Entier récupéré par le processus père

- Si le fils s'est terminé accidentellement à cause d'un signal, l'entier récupéré par le processus père contient le n° du signal

`WIFEXITED(status)`	Renvoie une valeur non nulle si le fils s'est terminé normalement
`WEXITSTATUS(status)`	Renvoie le code de retour du processus si le processus s'est terminé normalement
`WIFSIGNALED(status)`	Renvoie une valeur non nulle si le fils s'est terminé à cause d'un signal
`WTERMSIG(status)`	Renvoie le n° du signal qui a provoqué la mort du processus fils
`WIFSTOPPED(status)`	Valeur non nulle si le fils est stoppé
`WSTOPSIG(status)`	Renvoie le n° du signal qui a stoppé le processus fils

Synchronisation père-fils (Fin)



The Magic Garden Explained, The Internals of UNIX® System V Release 4, An Open Systems Design, Berny Goodheart & James Cox, Prentice Hall

Le recouvrement de programme

Cas d'utilisation du `fork`

Le `fork` est utilisé pour :

1. Permettre au père et au fils d'exécuter chacun et en concurrence *une partie différente du même programme*
   - Cas d'un serveur
2. Permettre au fils d'exécuter *un programme différent du père* en concurrence avec son père
   - Cas du `shell`

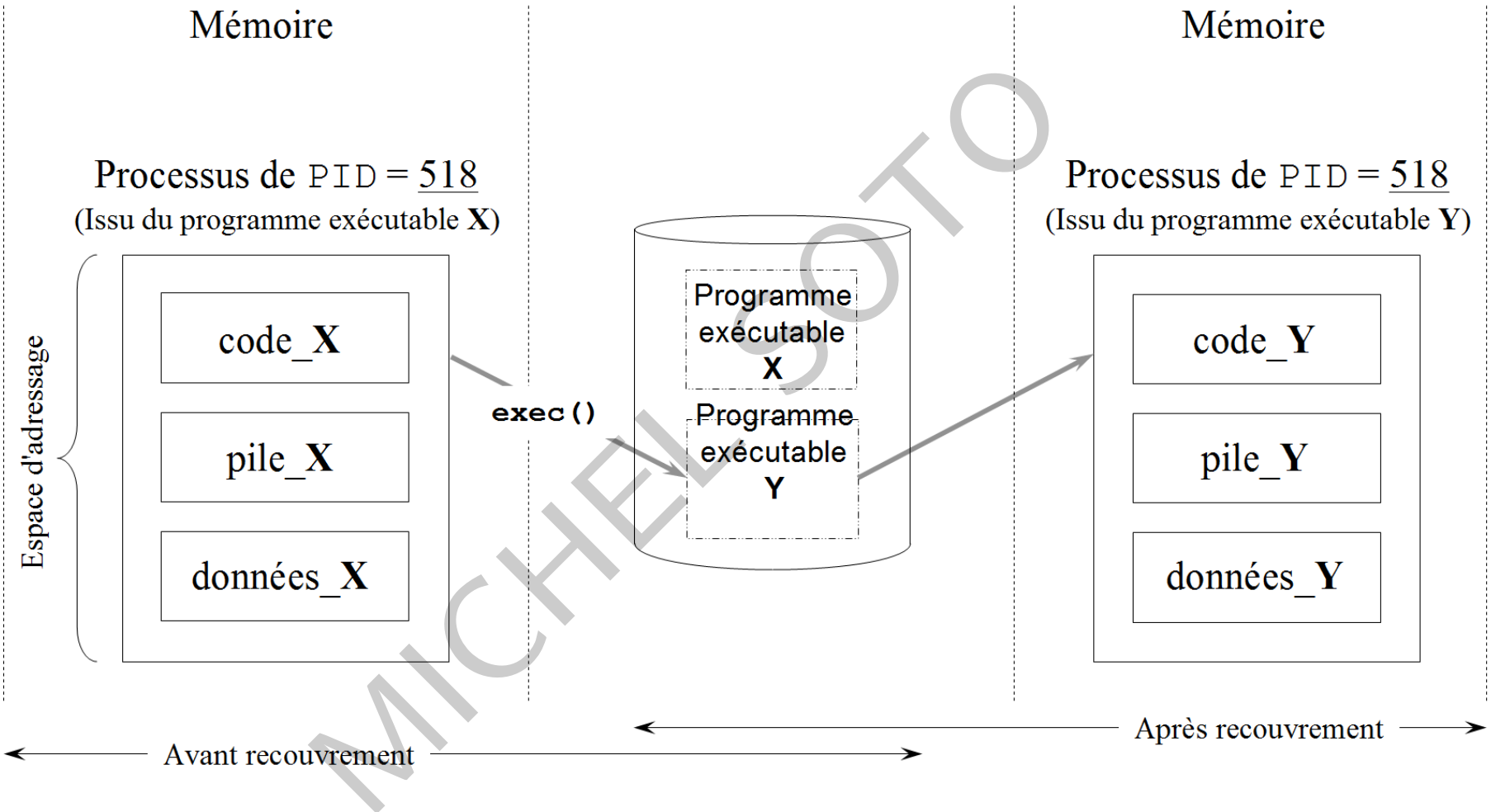
RECOUVREMENT

Dans le 2^{e}cas, ce sont les primitives de la famille `exec` qui vont permettre au processus fils de *recouvrir (écraser) intégralement* son code exécutable avec un autre code exécutable

REMARQUE

Il n'est pas nécessaire d'effectuer un `fork` avant de réaliser un `exec`

Le recouvrement de programme



Le recouvrement de programme (Suite)

```
#include <unistd.h>
int execl  (const char *path, const char *arg0, ... /* (char *)0 */);
int execlp (const char *file, const char *arg0, ... /* (char *)0 */);
int execle (const char *path, const char *arg0, ... /* (char *)0 */,
                                                    char *const envp[]);
int execv  (const char *path, char *const argv[]);
int execvp (const char *file, char *const argv[]);
int execve (const char *path, char *const argv[], char *const envp[]);
```

- Le premier argument doit pointer sur le nom du fichier associé au programme à exécuter
- Renvoient -1 an cas d'échec.

ATTENTION

*Ces primitives **ne retournent pas en cas de succès** mais uniquement en cas d'échec*

Le recouvrement de programme (Suite)

```
#include <unistd.h>
int execl  (const char *path, const char *arg0, ... /* (char *)0 */);
int execlp (const char *file, const char *arg0, ... /* (char *)0 */);
int execle (const char *path, const char *arg0, ... /* (char *)0 */,
                                                    char *const envp[]);
int execv  (const char *path, char *const argv[]);
int execvp (const char *file, char *const argv[]);
int execve (const char *path, char *const argv[], char *const envp[]);
```

- La transmission des arguments se fait soit :
  - par liste (suffixes `l`)
  - par un tableau de pointeur sur des chaines de caractères (suffixes `v`)
- Le fichier à exécuter est soit :
  - recherché en utilisant la variable d'environnement `PATH` (suffixe `p`)
  - indiqué dans le paramètre `path` (absence de suffixe `p`)
- L'environnement peut :
  - être modifié (suffixe `e`)
  - être conservé (absence de suffixe `e`)

Le recouvrement de programme (Suite)

Exemple: un shell embryonnaire

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define NB_ARGUMENTS 16

int main(int argc, char **argv){
char *argv_exec[NB_ARGUMENTS]; int indice;

 if (argc < 2) {printf("Usage : programme commande liste_arguments\n"); exit(1);}
 for (indice = 0; indice < argc; indice++) argv_exec[indice] = argv[indice + 1];
 argv_exec[indice]=NULL;
 if (execvp(argv_exec[0],argv_exec) == -1) {perror(argv_exec[0]); exit(2);}
}
```


REMARQUE

Le second if est inutile et on peut écrire:

```
execvp(argv_exec[0], argv_exec);
perror(argv_exec[0]);
exit(2);
```

puisque exec ne retourne qu'en cas d'échec

Exemple: un shell embryonnaire (Fin)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define NB_ARGUMENTS 16

int main(int argc, char **argv){
char *argv_exec[NB_ARGUMENTS]; int indice;

 if (argc < 2) {printf("Usage : programme commande liste_arguments\n"); exit(1);}
 for (indice = 0; indice < argc; indice++) argv_exec[indice] = argv[indice + 1];
 argv_exec[indice]=NULL;

 execvp(argv_exec[0],argv_exec);
 perror(argv_exec[0]);
 exit(2);
}
```

```
 >a.out ps -a
PID TTY TIME CMD
13458 pts/4 00:00:00 ps
29839 pts/0 00:00:00 tail
```

```
>a.out inexistant
inexistant: No such file or directory
```

Le recouvrement de programme (Fin)

Exemple: un shell un peu plus évolué

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
main() {char cmd[80]; int etatfils, i, pid;
printf(">");
while (fgets(cmd, 79, stdin) != NULL) {
        if ((pid=fork()) != 0) { // Père
                for (i=0; cmd[i]!='\n' && cmd[i]!='&'; i++);
                if (cmd[i] != '&') {wait(&etatfils);
                                printf("Code de retour = %d\n", WEXITSTATUS(etatfils));
                }
                else printf("[%d]\n", pid);
                printf(">");
        }
        else {// Fils
                  for (i=0; cmd[i]!='\n' && cmd[i]!='&'; i++);
                  cmd[i] = '\0';
                  execlp(cmd, cmd, NULL);
                  perror("Erreur a l'execl");
                  exit(1);
                }
        }// while
}// main
```