

PLAN

- 1 Caractéristiques
-
- 2 Primitives

GESTION DES PROCESSUS

2

K. ElBedoui

Chpitre 2. Processus

1

K. ElBedoui

Chpitre 2. Processus

CARACTÉRISTIQUES

- 1. Structure en mémoire



Instructions en assembleur

Code

Données

Var, allocation dynamique de la mémoire

Pile

Les appels aux fonctions

Chpitre 2. Processus

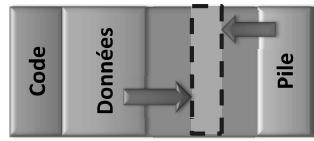
K. ElBedoui

Chpitre 2. Processus

4

1 CARACTÉRISTIQUES

1. Structure en mémoire



Les deux segments de données et de pile :

- variant dans un sens inverse
- admettent une limite

1 CARACTÉRISTIQUES

2. Structure de contrôle

Le processus est géré via une structure de donnée dite :
PCB (Process Control Bloc) qui est une structure de données.

6

K. ElBedoui

Chpitre 2. Processus

1

1 CARACTÉRISTIQUES

2. Structure de contrôle

Table de processus

PCB 1	PCB2	PCBn
PID	L'ID du processus	L'ID de son processus père

Le pointeur d'instruction
Actif, prêt, bloqué,...

Chpitre 2. Processus

K. ElBedoui

8

2 PRIMITIVES DE GESTION DES PROCESSUS

Les opérations possibles sur un processus :

1. Crédit
2. Terminaison
3. Exécution
4. Mise en attente et réveil
5. Modification de la priorité

Chpitre 2. Processus

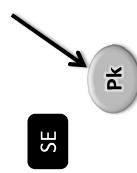
K. ElBedoui

7

2 PRIMITIVES DE GESTION DES PROCESSUS

1. Création d'un processus

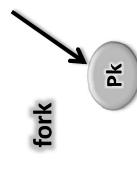
La création d'un processus nécessite un appel système



2 PRIMITIVES DE GESTION DES PROCESSUS

1. Création d'un processus

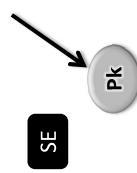
La création d'un processus nécessite un appel système



2 PRIMITIVES DE GESTION DES PROCESSUS

1. Création d'un processus

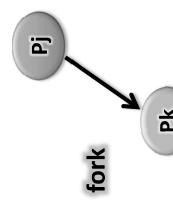
La création d'un processus nécessite un appel système



2 PRIMITIVES DE GESTION DES PROCESSUS

1. Création d'un processus

La création d'un processus nécessite un appel système

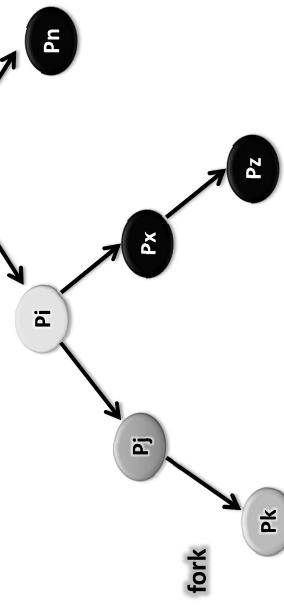


Le processus créateur s'appelle Père
Le processus créé s'appelle Fils

2 PRIMITIVES DE GESTION DES PROCESSUS

1. Création d'un processus

Racine = init
PID = 1



La commande shell `psTree` permet d'afficher cette arborescence de processus

PRIMITIVES DE GESTION DES PROCESSUS

2

PRIMITIVES DE GESTION DES PROCESSUS

```
Père          1. Création d'un processus
              #include <unistd.h>
              void main()
              {
                  int p;
                  p=fork();
              }
              Fils
```

Chpitre 2. Processus

17

K. ElBedoui

1. Création d'un processus

```
# include <unistd.h>
void main()
{
    int p;
    p=fork();
```

■ Le père connaît :

- Son propre identificateur (PID)
- L'identificateur de son fils créé ???

18

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

2

PRIMITIVES DE GESTION DES PROCESSUS

```
1. Création d'un processus
# include <unistd.h>
void main()
{
    int p;
    p=fork();
    Switch (p) :
    { case -1:
        case 0:
    default:
```

■ Le fils connaît :

- Son propre identificateur (PID)
- L'identificateur de son père (PPID)

fork

Chpitre 2. Processus

19

K. ElBedoui

1. Création d'un processus

```
# include <unistd.h>
void main()
{
    int p;
    p=fork();
    Switch (p) :
    { case -1:
        case 0:
    default:
```

fork

20

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

2 PRIMITIVES DE GESTION DES PROCESSUS

```
1. Création d'un processus
#include <unistd.h>
void main()
{
    int p;
    p=fork();
    Switch (p) :
    { case -1 : /*erreur de création */
    case 0 : /* partie du fils ; travail accordé au fils*/
    default :
    }
}
```

21

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

```
1. Création d'un processus
#include <unistd.h>
void main()
{
    int p;
    p=fork();
    Switch (p) :
    { case -1 : /*erreur de création */
    case 0 : /* partie du fils ; travail accordé au fils*/
    /* partie du père : la valeur de retour de fork est
    le PID du processus fils créé*/
    default :
    }
}
```

23

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

```
1. Création d'un processus
#include <unistd.h>
void main()
{
    int p;
    p=fork();
    Switch (p) :
    { case -1 :
    case 0 :
    default :
    }
}
```

24

K. ElBedoui

2 PRIMITIVES DE GESTION DES PROCESSUS

2 PRIMITIVES DE GESTION DES PROCESSUS

```
1. Création d'un processus
Père #include <unistd.h>
void main()
{
    int p;
    p=fork();
    Fils
    Switch (p) :
    { case -1:
        case 0:
        default:
    }
}
```

Chapitre 2. Processus

25

K. ElBedoui

```
1. Création d'un processus
Père #include <unistd.h>
void main()
{
    int p;
    p=fork();
    Fils
    Switch (p) :
    { case -1:
        case 0:
        default:
    }
}
```

Chapitre 2. Processus

26

K. ElBedoui

2 PRIMITIVES DE GESTION DES PROCESSUS

2

```
1. Création d'un processus
Père #include <unistd.h>
void main()
{
    int p;
    p=fork();
    Fils
    Switch (p) :
    { case -1:
        case 0:
        default:
    }
}
```

Chapitre 2. Processus

27

K. ElBedoui

```
1. Création d'un processus
Prog1 int main(void)
{
    int p;
    switch(p=fork())
    {
        case -1:
            perror("Erreur de création d'un processus");
            exit(-1);
        case 0:
            printf("Je suis le processus fils ! Mon PID est %d\n", getpid());
            exit(0);
        default:
            printf("Je suis le processus père ! Mon PID est %d\n", getpid());
            exit(0);
    }
    return 0;
}
```

Chapitre 2. Processus

28

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

2

PRIMITIVES DE GESTION DES PROCESSUS

Exemple 1

1. **Création d'un processus**

```
Prog1 int main(void)
{
    int p;
    switch(p=fork())
    {
        case -1:
            perror("Erreur de création d'un processus");
            exit(-1);
        case 0:
            printf("Je suis le processus fils ! Mon PID est %d\n", getpid());
            exit(0);
        default:
            printf("Je suis le processus père ! Mon PID est %d\n", getpid());
            exit(0);
    }
    return 0;
}
```

Chapitre 2. Processus

29

K. ElBedoui

Exemple 1

1. **Création d'un processus**

Prog2

Chaque processus affiche son PID via la primitive getpid()

```
./Prog1
Je suis le processus père ! Mon PID est 14582
Je suis le processus fils ! Mon PID est 15200
```

Première exécution

30

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

2

PRIMITIVES DE GESTION DES PROCESSUS

Exemple 1

1. **Création d'un processus**

Prog1

Deuxième exécution

```
./Prog1
Je suis le processus père ! Mon PID est 14582
Je suis le processus fils ! Mon PID est 15200
./Prog1
Je suis le processus fils ! Mon PID est 15356
Je suis le processus père ! Mon PID est 15250
```

Deuxième exécution

./Prog1
Je suis le processus père ! Mon PID est 14582

Je suis le processus fils ! Mon PID est 15200

./Prog1

Je suis le processus fils ! Mon PID est 15356

Je suis le processus père ! Mon PID est 15250

L'ordre de passage entre le père et

le fils sur le processus dépend de l'algorithme de l'ordonnancement.

À chaque lancement du programme, il y aura création d'un nouveau père et fils.

Chapitre 2. Processus

31

K. ElBedoui

32

PRIMITIVES DE GESTION DES PROCESSUS

2

PRIMITIVES DE GESTION DES PROCESSUS

Exemple 2

```
1. Création d'un processus  
Prog2 int main(void)  
{  
    int p;  
    switch(p=fork())  
    {  
        case -1:  
            perror("Erreur de création d'un processus");  
            exit(-1);  
  
        case 0:  
            printf("Je suis le processus fils ! Mon PID est %d\n", getpid());  
            printf("Le PID de mon père est : %d\n", p);  
            exit(0);  
  
        default:  
            printf("Je suis le processus père ! Mon PID est %d\n", getpid());  
            printf("Le PID de mon fils est : %d\n", p);  
            exit(0);  
    }  
    return 0;  
}
```

Chpitre 2. Processus K. ElBedoui 33

Exemple 2

```
1. Création d'un processus  
Prog2 #include <sys/types.h>  
#include <sys/wait.h>  
#include <unistd.h>  
#include <stdio.h>  
#include <errno.h>  
#include <stropts.h>  
  
int main()  
{  
    pid_t p;  
    if ((p = fork()) < 0) // si l'erreur de fork  
    {  
        perror("Erreur de création d'un processus via fork()");  
        exit(1);  
    }  
  
    if (p == 0) // si le processus fils  
    {  
        // code à exécuter dans le fils  
        // ...  
        exit(0);  
    }  
    else // si le processus père  
    {  
        // code à exécuter dans le père  
        // ...  
        waitpid(p, NULL, 0);  
        exit(0);  
    }  
}
```

Chpitre 2. Processus K. ElBedoui 34

PRIMITIVES DE GESTION DES PROCESSUS

2

PRIMITIVES DE GESTION DES PROCESSUS

Exemple 2

```
1. Création d'un processus  
Prog2 #include <sys/types.h>  
#include <sys/wait.h>  
#include <unistd.h>  
#include <stropts.h>  
  
int main()  
{  
    pid_t p;  
    if ((p = fork()) < 0) // si l'erreur de fork  
    {  
        perror("Erreur de création d'un processus via fork()");  
        exit(1);  
    }  
  
    if (p == 0) // si le processus fils  
    {  
        // code à exécuter dans le fils  
        // ...  
        exit(0);  
    }  
    else // si le processus père  
    {  
        // code à exécuter dans le père  
        // ...  
        waitpid(p, NULL, 0);  
        exit(0);  
    }  
}
```

Exemple 2

```
1. Création d'un processus  
Prog2 #include <sys/types.h>  
#include <sys/wait.h>  
#include <unistd.h>  
#include <stropts.h>  
  
int main()  
{  
    pid_t p;  
    if ((p = fork()) < 0) // si l'erreur de fork  
    {  
        perror("Erreur de création d'un processus via fork()");  
        exit(1);  
    }  
  
    if (p == 0) // si le processus fils  
    {  
        // code à exécuter dans le fils  
        // ...  
        exit(0);  
    }  
    else // si le processus père  
    {  
        // code à exécuter dans le père  
        // ...  
        waitpid(p, NULL, 0);  
        exit(0);  
    }  
}
```

Chpitre 2. Processus K. ElBedoui 35

PRIMITIVES DE GESTION DES PROCESSUS

2

PRIMITIVES DE GESTION DES PROCESSUS

Exemple 2

```
1. Création d'un processus  
Prog2 #include <sys/types.h>  
#include <sys/wait.h>  
#include <unistd.h>  
#include <stropts.h>  
  
int main()  
{  
    pid_t p;  
    if ((p = fork()) < 0) // si l'erreur de fork  
    {  
        perror("Erreur de création d'un processus via fork()");  
        exit(1);  
    }  
  
    if (p == 0) // si le processus fils  
    {  
        // code à exécuter dans le fils  
        // ...  
        exit(0);  
    }  
    else // si le processus père  
    {  
        // code à exécuter dans le père  
        // ...  
        waitpid(p, NULL, 0);  
        exit(0);  
    }  
}
```

Exemple 2

```
1. Création d'un processus  
Prog2 #include <sys/types.h>  
#include <sys/wait.h>  
#include <unistd.h>  
#include <stropts.h>  
  
int main()  
{  
    pid_t p;  
    if ((p = fork()) < 0) // si l'erreur de fork  
    {  
        perror("Erreur de création d'un processus via fork()");  
        exit(1);  
    }  
  
    if (p == 0) // si le processus fils  
    {  
        // code à exécuter dans le fils  
        // ...  
        exit(0);  
    }  
    else // si le processus père  
    {  
        // code à exécuter dans le père  
        // ...  
        waitpid(p, NULL, 0);  
        exit(0);  
    }  
}
```

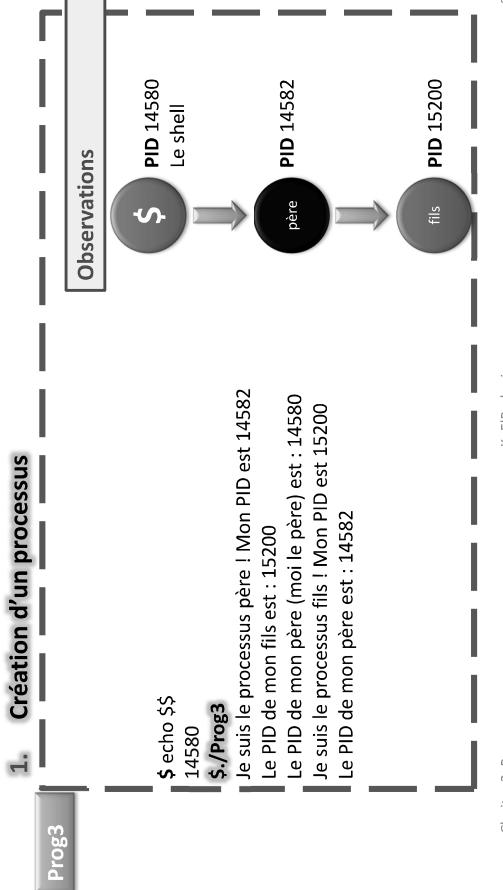
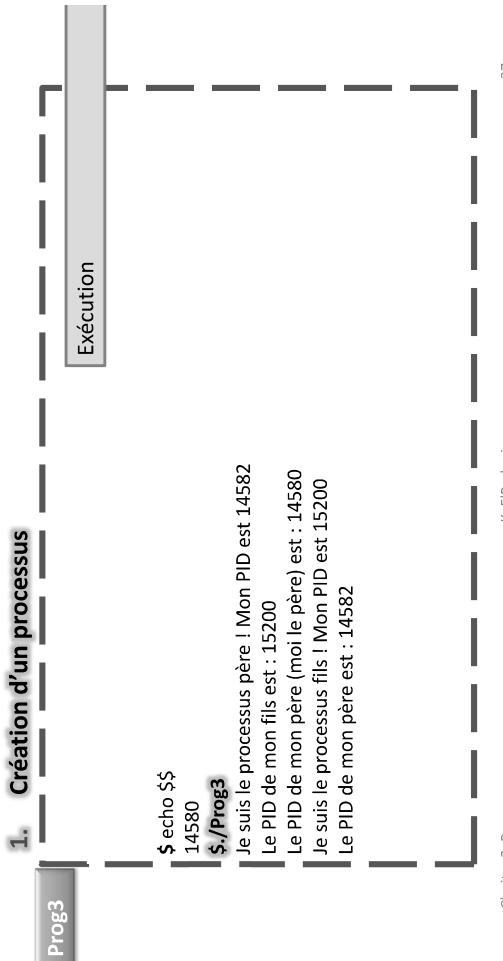
Chpitre 2. Processus K. ElBedoui 36

PRIMITIVES DE GESTION DES PROCESSUS

2

PRIMITIVES DE GESTION DES PROCESSUS

Exemple 1



PRIMITIVES DE GESTION DES PROCESSUS

2

PRIMITIVES DE GESTION DES PROCESSUS

2. Terminaison d'un processus

❖ Une terminaison normale

La terminaison d'un processus du système peut être :

❖ Une terminaison normale : s'il a achevé ses instructions et a réussi à effectuer son travail.

❖ Une terminaison anormale : si un dysfonctionnement est découvert au point qu'il ne permet pas au programme de continuer son travail.

▪ Selon plusieurs manière : la plus simple est de revenir de la fonction main() en renvoyant une valeur entière.

▪ Cette valeur est lue par le processus père, qui peut en tirer les conséquences adéquates.

▪ Par convention, un programme qui réussit à effectuer son travail renvoie une valeur nulle. Tandis que les autres cas sont indiqués par des codes de retour non nuls (et qui peuvent être documentés avec l'application).

▪ Dans la plupart des cas, on ne teste que la nullité du code de retour.

▪ Lorsque le processus est arrêté à cause d'un signal, le shell modifie le code de retour (bash ajoute 128, par exemple). Il est donc conseillé de n'utiliser que des valeurs comprises entre 0 et 127.

PRIMITIVES DE GESTION DES PROCESSUS

PRIMITIVES DE GESTION DES PROCESSUS

2

2. Terminaison d'un processus

❖ Une terminaison normale

- Lorsqu'un processus lancé par le shell se termine, son code de retour est disponible dans la variable spéciale \$?.

```
$ grep root /etc/passwd
root:x:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
$ echo $?
0
```

❖ Le code de retour est nul, indiquant la réussite.

Chpitre 2. Processus

41

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

2

2. Terminaison d'un processus

❖ Une terminaison normale

- Lorsqu'un processus lancé par le shell se termine, son code de retour est disponible dans la variable spéciale \$?.

```
$ grep root /etc/inexisting
grep: /etc/inexisting: Aucun fichier ou répertoire de ce type
$ echo $?
2
```

❖ La valeur 2 signifie la demande est invalide.

Chpitre 2. Processus

43

K. ElBedoui

2. Terminaison d'un processus

❖ Une terminaison normale

- Lorsqu'un processus lancé par le shell se termine, son code de retour est disponible dans la variable spéciale \$?.

```
$ grep abcdefg /etc/passwd
$ echo $?
1
```

❖ La valeur 1 signifie que le travail est fait correctement, mais la chaîne n'est pas trouvée.

Chpitre 2. Processus

42

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

2

2. Terminaison d'un processus

❖ Une terminaison normale

- Lorsqu'un processus lancé par le shell se termine, son code de retour est disponible dans la variable spéciale \$?.

```
$ grep
$ echo $?
$ 0,1,2...
$
```

Tous ces codes (documentés dans la page de manuel de grep), sont renvoyés par le processus quand il se termine normalement – de son plein gré.

Chpitre 2. Processus

44

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

PRIMITIVES DE GESTION DES PROCESSUS

2

2. Terminaison d'un processus

❖ Une terminaison normale

- Lorsqu'un processus lancé par le shell se termine, son code de retour est disponible dans la variable spéciale \$?.

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
\$ sleep (30) (Ctrl ^ C)	2	Term	Interrupt from keyboard
\$ echo \$?	3	Core	Quit from keyboard
	4	Core	Illegal Instruction
	6	Core	Abort signal from abort(3)
	8	Core	Floating point exception
	9	Term	Kill signal
	11	Core	Invalid memory reference
	13	Term	Broken pipe: write to pipe with no readers
	14	Term	Timer signal from alarm(2)
SIGALRM	15	Term	Termination signal
SIGTERM			

Chpitre 2. Processus

45

- Lorsqu'un processus lancé par le shell se termine, son code de retour est disponible dans la variable spéciale \$?.

\$ sleep (30)	\$ sleep (30)	(Ctrl ^ C)
\$ echo \$?	\$ echo \$?	
	130	
	\$	

K. ElBedoui

46

PRIMITIVES DE GESTION DES PROCESSUS

PRIMITIVES DE GESTION DES PROCESSUS

2

2. Terminaison d'un processus

❖ Une terminaison normale

- Si seuls la réussite ou l'échec du programme importent (si le processus père n'essaye pas de détailler les raisons de l'échec), il est possible d'employer les constantes symboliques EXIT_SUCCESS ou EXIT_FAILURE définies dans <stdlib.h>.

- On lui transmet en argument le code de retour pour le processus père. L'effet est strictement égal à celui d'un retour depuis la fonction main(), à la différence que exit() peut être invoquée depuis n'importe quelle partie du programme (notamment depuis les routines de traitement d'erreur).

Chpitre 2. Processus

47

K. ElBedoui

K. ElBedoui

48

PRIMITIVES DE GESTION DES PROCESSUS

PRIMITIVES DE GESTION DES PROCESSUS

2

2. Terminaison d'un processus

- ❖ Une terminaison anormale

- Un programme peut également se terminer de manière anormale.
- Ceci est le cas par exemple lorsqu'un processus exécute une instruction illégale, ou qu'il essaye d'accéder au contenu d'un pointeur mal initialisé.
- Ces actions déclenchent un signal qui, par défaut, arrête le processus en créant un fichier d'image mémoire core.

Chapitre 2. Processus

49

K. ElBedoui

2. Terminaison d'un processus

- ❖ Une terminaison anormale

- Une manière "propre" d'interrompre abnormalment un programme (par exemple lorsqu'un bogue est découvert) est d'invoquer la fonction `abort()`.
- Le problème de la fonction `abort()` ou des arrêts dus à des signaux est qu'il est difficile de déterminer ensuite à quel endroit du programme le dysfonctionnement a eu lieu.
- Cependant, il est possible d'autopsier le fichier core. Chose qui est parfois ardue.

50

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

PRIMITIVES DE GESTION DES PROCESSUS

2

2. Terminaison d'un processus

- ❖ Une terminaison anormale

- Une autre manière de détecter automatiquement les bogues est d'utiliser systématiquement la fonction `assert()` dans les parties critiques du programme.
- Il s'agit d'une macro, définie dans <assert.h>, qui évalue l'expression qu'on lui transmet en argument:
 - Si l'expression est vraie, elle ne fait rien.
 - Par contre, si elle est fausse, `assert()` arrête le programme après avoir écrit un message sur la sortie d'erreur standard indiquant :
- Le fichier source concerné,
- La ligne de code
- et le texte de l'assertion ayant échoué.

Il est alors très facile de se reporter au point décrit pour rechercher le bogue.

Chapitre 2. Processus

51

K. ElBedoui

2. Terminaison d'un processus

- ❖ Une terminaison anormale

```
int main (void)
{
    fonction_reussissant(5);
    fonction_reussissant(5);
    fonction_ecchouant(5);
    return EXIT_SUCCESS;
}

void fonction_reussissant (int i)
{
    /* Cette fonction nécessite que i soit positif */
    assert(i > 0);
    fprintf(stderr, "Ok, i est positif \n");
}

void fonction_ecchouant (int i)
{
    /* Cette fonction nécessite que i soit négatif */
    assert(i <= 0);
    fprintf(stderr, "Ok, i est négatif \n");
}
```

52

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

2 PRIMITIVES DE GESTION DES PROCESSUS

2. Terminaison d'un processus

❖ Une terminaison anormale

```
int main (void)
{
    fonction_reussissant(5);
    fonction_échouant(5);
    return EXIT_SUCCESS;
}

void fonction_reussissant (int i)
{
    /* Cette fonction nécessite que i soit positif */
    assert(i >= 0);
    fprintf(stdout, "Ok, i est positif \n");
}

void fonction_échouant (int i)
{
    /* Cette fonction nécessite que i soit négatif */
    assert(i <= 0);
    fprintf(stdout, "Ok, i est négatif \n");
}
```

Chapitre 2. Processus

53

2. Terminaison d'un processus

❖ Une terminaison normale

```
int main (void)
{
    fonction_reussissant(5);
    fonction_échouant(5);
    return EXIT_SUCCESS;
}

void fonction_reussissant (int i)
{
    /* Cette fonction nécessite que i soit positif */
    assert(i >= 0);
    fprintf(stdout, "Ok, i est positif \n");
}

void fonction_échouant (int i)
{
    /* Cette fonction nécessite que i soit négatif */
    assert(i <= 0);
    fprintf(stdout, "Ok, i est négatif \n");
}
```

K. ElBedoui

54

Lors de l'exécution, la première assertion passe, et le message est écrit sur l'écran, mais la seconde assertion échoue et assert() affiche alors le détail du problème sur l'écran.

/* Cette fonction nécessite que i soit négatif */

assert(i <= 0);

fprintf(stdout, "Ok, i est positif \n");

*/

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

2

PRIMITIVES DE GESTION DES PROCESSUS

2. Terminaison d'un processus

❖ Attendre la fin d'un processus

- Un processus qui se termine passe automatiquement par un état spécial, zombie, en attendant que le processus père ait lu son code de retour.

- Si le processus père ne lit pas le code de retour de son fils, celui-ci peut rester indéfiniment à l'état zombie.

2. Terminaison d'un processus

❖ Attendre la fin d'un processus

- ❖ Un processus fils se trouve donc dans l'état Zombie s'il termine son exécution (il ne consomme plus du temps processeur) mais il est encore représenté par un PCB.

- ❖ Un processus fils défunt reste Zombie jusqu'à ce que son père soit informé de sa mort ensuite il sera supprimé.

- ❖ Si le processus fils est orphelin (suppression de son père), alors il sera adopté par le processus init (dont le PID est 1).

PRIMITIVES DE GESTION DES PROCESSUS

2 PRIMITIVES DE GESTION DES PROCESSUS

2. Terminaison d'un processus
 - ❖ Attendre la fin d'un processus

état	Anglais	Signification
Exécution	Running (R)	Le processus est en cours de fonctionnement, il effectue un travail actif.
Sommeil	Sleeping (S)	Le processus est en attente d'un événement extérieur. Il se met en sommeil.
Arrêt	Stopped (T)	Le processus a été temporairement arrêté par un signal. Il ne s'exécute plus et ne réagira qu'à un signal de redémarrage.
Zombie	Zombie (Z)	Le processus s'est terminé, mais son père n'a pas encore lu son code de retour.

Chapitre 2. Processus 57 K. ElBedoui

2. Terminaison d'un processus
 - ❖ Attendre la fin d'un processus

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

void main()
{
    if (fork () ==0)
    {
        printf(" Fin du processus fils de PID %d\n", getpid());
        exit (2);
    }
    sleep (30);
}
```

Chapitre 2. Processus 58 K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

2 PRIMITIVES DE GESTION DES PROCESSUS

2. Terminaison d'un processus
 - ❖ Attendre la fin d'un processus

```
[user@localhost ~]$ ps -ax
1737 ? S 0:00 gnome-pty-helper
1738 pts/0 Ss 0:00 bash
1770 pts/0 St 0:00 /a.out
1771 pts/0 Z+ 0:00 ./a.out <defunct>
1772 pts/1 Ss 0:00 bash
1794 pts/1 R+ 0:00 ps -ax
[user@localhost ~]$
```

Chapitre 2. Processus K. ElBedoui

Exemple

Exemple

60

```
[user@localhost ~]$ ps -ax
1737 ? S 0:00 gnome-pty-helper
1738 pts/0 Ss 0:00 bash
1770 pts/0 St 0:00 /a.out
1771 pts/0 Z+ 0:00 ./a.out <defunct>
1772 pts/1 Ss 0:00 bash
1794 pts/1 R+ 0:00 ps -ax
[user@localhost ~]$
```

Chapitre 2. Processus K. ElBedoui

Chapitre 2. Processus

59

PRIMITIVES DE GESTION DES PROCESSUS

PRIMITIVES DE GESTION DES PROCESSUS

2

2. Terminaison d'un processus

❖ Attendre la fin d'un processus

```
[user@localhost ~]$ ./a.out  
Fin du processus fils e PID : 1771  
[user@localhost ~]$ █
```

Exemple

2. Terminaison d'un processus

❖ Attendre la fin d'un processus

- Pour éviter ce problème de processus zombie et assurer une fin propre; il faut assurer la synchronisation entre le processus père et son fils et ce via les primitives wait

```
[user@localhost ~]$ ps -ax  
1737 ? S 0:00 gnome-pty-helper  
1738 pts/0 Ss+ 0:00 bash  
1772 pts/1 R 0:00 bash  
1796 pts/1 R+ 0:00 ps -ax  
[user@localhost ~]$ █
```

Chpitre 2. Processus

61

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

2

2. Terminaison d'un processus

❖ Attendre la fin d'un processus

Les primitives wait

Bibliothèques <sys/wait.h>

- ❖ Le père est suspendu jusqu'à ce qu'un de ses fils se termine.
- ❖ Si le fils est déjà Zombie au moment de l'appel alors wait() renvoie immédiatement le résultat : PID du fils et le code de retour de celui-ci dans la variable status.

`pid_t wait(int *status);`

Types :

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Chpitre 2. Processus

63

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

2

2. Terminaison d'un processus

❖ Attendre la fin d'un processus

- Un processus peut se mettre en attente de la mort de l'un de ses fils via wait() ou d'une manière précise avec waitpid()
- Lorsque le père est informé de la mort de l'un de ses fils alors il procède à la destruction de son PCB (celui du fils)

62

K. ElBedoui

64

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

PRIMITIVES DE GESTION DES PROCESSUS

2

- 2. Terminaison d'un processus
 - ❖ Attendre la fin d'un processus

```
pid_t waitpid (pid_t pid, int *status, int options);
```

- ❖ L'identité du processus attendu est spécifiée dans la variable pid.
- ❖ Le contenu de la variable pid est interprété :
 - Si >0 alors suspension du père jusqu'à ce que le processus de PID pid se termine
 - Si =0 alors suspension du père jusqu'à ce que n'importe quel fils (é même groupe que le père) meure.
 - Si =-1 alors suspension du père jusqu'à ce que n'importe quel fils meure.

Chpitre 2. Processus

65

- 2. Terminaison d'un processus
 - ❖ Attendre la fin d'un processus

```
pid_t waitpid (pid_t pid, int *status, int options);
```

- ❖ Le contenu de la variable status est interprété :
 - WIFEXITED(status) : Vrai si le fils s'est terminé par un appel à exit()
 - WEXITSTATUS(status) : permet de récupérer le code passé par le fils au moment de sa terminaison
 - WIFSIGNALED(status) : Permet de savoir que le fils s'est terminé à cause d'un signal
 - WIFSTOPPED(status) : Indique que le fils est arrêté temporairement.

K. ElBedoui

66

PRIMITIVES DE GESTION DES PROCESSUS

2

- 2. Terminaison d'un processus
 - ❖ Attendre la fin d'un processus

```
pid_t waitpid (pid_t pid, int *status, int options);
```

- ❖ Le champ options peut prendre la valeur WNOHANG qui provoque un retour immédiat si aucun fils ne s'est encore terminé.
- ❖ En cas de succès, la primitive renvoie le pid du processus fils qui s'est terminé
- ❖ La primitive renvoie -1 en cas d'échec (la variable errno est positionnée)
- ❖ Si WNOHANG a été utilisée et qu'aucun fils ne s'est terminé, la primitive renvoie 0.

Chpitre 2. Processus

67

PRIMITIVES DE GESTION DES PROCESSUS

2

- 3. Exécution d'un processus

- Pour récupérer les arguments dans le programme C, on utilise les paramètres argc et argv du main.

```
int main (int argc, char *argv[ ])
```
- L'entier argc donne le nombre d'arguments rentrés dans la ligne de commande plus 1
- Le paramètre argv est un tableau de chaînes de caractères qui contient comme éléments :
 - Le premier élément argv[0] est une chaîne qui contient le nom du fichier exécutable du programme ;
 - Les éléments suivants argv[1], argv[2],etc... sont des chaînes de caractères qui contiennent les arguments passés en ligne de commande.

K. ElBedoui

68

PRIMITIVES DE GESTION DES PROCESSUS

2

PRIMITIVES DE GESTION DES PROCESSUS

3. Exécution d'un processus

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    if (argc == 1)
        puts("Le programme n'a reçu aucun argument");
    if (argc >= 2)
    {
        puts("Le programme a reçu les arguments suivants :");
        for (i=1 ; i<argc ; i++)
            printf("Argument %d = %s\n", i, argv[i]);
    }
    return 0;
}
```

Chpitre 2. Processus

69

K. ElBedoui

70

PRIMITIVES DE GESTION DES PROCESSUS

2

PRIMITIVES DE GESTION DES PROCESSUS

3. Exécution d'un processus

Les primitives exec :

- **int exec (const char * app, const char * arg, ...);**
app : chemin complet de l'application
arg : paramètres sous forme d'une liste d'argument, termine par pointeur NULL.
- **int execv (const char * app, const char * argv[]);**
app : chemin complet de l'application
argv[] : paramètres à donner à l'application sous la forme d'un vecteur (tableau)

Chpitre 2. Processus

71

Chpitre 2. Processus

72

K. ElBedoui

2 PRIMITIVES DE GESTION DES PROCESSUS

2 PRIMITIVES DE GESTION DES PROCESSUS

3. Exécution d'un processus

Exemple 1

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

Le premier paramètre est une chaîne qui doit
contenir le chemin d'accès complet (dans le
système de fichiers) au fichier exécutables ou
au script shell à exécuter.

int main()
{
    /* dernier élément NULL, OBLIGATOIRE */
    exec("./usr/bin/emacs", "emacs", "fichier.c", "fichier.h", NULL);

    perror("Problème : cette partie du code ne doit jamais être exécutée");
    return 0;
}
```

Chapitre 2. Processus

77

K. ElBedoui

3. Exécution d'un processus

Exemple 1

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

Les paramètres suivants sont les argv. La chaîne
argv[0] doit donner le nom du programme (sans
chemin d'accès), et les haînes suivantes argv[1],
argv[2],etc... donnent les arguments.

int main()
{
    /* dernier élément NULL, OBLIGATOIRE */
    exec("./usr/bin/emacs", "emacs", "fichier.c", "fichier.h", NULL);

    perror("Problème : cette partie du code ne doit jamais être exécutée");
    return 0;
}
```

Chapitre 2. Processus

78

K. ElBedoui

2 PRIMITIVES DE GESTION DES PROCESSUS

2 PRIMITIVES DE GESTION DES PROCESSUS

3. Exécution d'un processus

Exemple 1

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

La partie qui suit l'appel d'exec n'est jamais
exécutée

int main()
{
    /* dernier élément NULL, OBLIGATOIRE */
    exec("./usr/bin/emacs", "emacs", "fichier.c", "fichier.h", NULL);

    perror("Problème : cette partie du code ne doit jamais être exécutée");
    return 0;
}
```

Chapitre 2. Processus

79

K. ElBedoui

2 PRIMITIVES DE GESTION DES PROCESSUS

2 PRIMITIVES DE GESTION DES PROCESSUS

3. Exécution d'un processus

Exemple 2

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

La fonction execvp permet de rechercher les
exécutables dans les répertoires apparaissant
dans le PATH, ce qui évite souvent d'avoir à
spécifier le chemin complet.

int main()
{
    /* dernier élément NULL, OBLIGATOIRE */
    execvp("emacs", "emacs", "fichier.c", "fichier.h", NULL);

    perror("Problème : cette partie du code ne doit jamais être exécutée");
    return 0;
}
```

Chapitre 2. Processus

80

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

PRIMITIVES DE GESTION DES PROCESSUS

2

3. Exécution d'un processus

La différence entre execv et execcl est que l'on n'a pas besoin de connaître la liste des arguments à l'avance (ni même leur nombre). Rappelons le prototype :

```
execv (const char* ref, const char* argv[]);
```

Le mot **const** signifie que la fonction execv ne modifie pas ses paramètres.

- Le **premier paramètre** est une chaîne qui doit contenir le chemin d'accès.
- Le **deuxième paramètre** est un tableau de chaînes de caractères donnant les arguments passés au programme à lancer dans un format similaire au paramètre argv du main de ce programme. La chaîne argv[0] doit donner le nom du programme (sans chemin d'accès), et les chaînes suivants argv[1], argv[2],etc... donnent les arguments.

Chapitre 2. Processus

81

K. ElBedoui

82

K. ElBedoui

83

Chapitre 2. Processus

84

K. ElBedoui

PRIMITIVES DE GESTION DES PROCESSUS

PRIMITIVES DE GESTION DES PROCESSUS

2

3. Exécution d'un processus

PRIMITIVES DE GESTION DES PROCESSUS

PRIMITIVES DE GESTION DES PROCESSUS

2

Exemple 3

2

3. Exécution d'un processus

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

int main()
{
    char * argv[] = {"emacs", "fichier.c", "fichier.h", NULL};
    /* dernier élément NULL, obligatoire */
    execv("/usr/bin/emacs", argv);
    puts("Problème : cette partie du code ne doit jamais être exécutée");
    return 0;
}
```

Chapitre 2. Processus

83

K. ElBedoui

84

K. ElBedoui

2 PRIMITIVES DE GESTION DES PROCESSUS

2 PRIMITIVES DE GESTION DES PROCESSUS

3. Exécution d'un processus

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main()
{
    pid_t pid;
    pid=fork();
    if(pid==1)
        printf("Problème de création de processus !");
    else if(pid==0)
    {
        printf("Je suis le fils, mon pid est %d\n", getpid());
        printf("Le pid de mon père est %d\n", getppid());
        execvp("ls","ls","-l","/",NULL);
    }
    else
    {
        printf("Je suis le père, mon pid est %d\n", getpid());
        printf("Le pid de mon fils est %d\n", pid);
        wait();
    }
    return 0;
}
```

Chapitre 2. Processus K. ElBedoui 85

3. Exécution d'un processus

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main()
{
    pid_t pid;
    pid=fork();
    if(pid==1)
        printf("Problème de création de processus !");
    else if(pid==0)
    {
        1 printf("Je suis le fils, mon pid est %d\n", getpid());
        2 printf("Le pid de mon père est %d\n", getppid());
        3 execvp("ls","ls","-l","/",NULL);
    }
    else
    {
        4 printf("Je suis le père, mon pid est %d\n", getpid());
        5 printf("Le pid de mon fils est %d\n", pid);
        wait();
    }
    return 0;
}
```

Chapitre 2. Processus K. ElBedoui 86

PRIMITIVES DE GESTION DES PROCESSUS

3. Exécution d'un processus

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main()
{
    pid_t pid;
    pid=fork();
    if(pid==1)
        printf("Problème de création de processus !");
    else if(pid==0)
    {
        1 printf("Je suis le fils, mon pid est %d\n", getpid());
        2 printf("Le pid de mon père est %d\n", getppid());
        3 execvp("ls","ls","-l","/",NULL);
    }
    else
    {
        4 printf("Je suis le père, mon pid est %d\n", getpid());
        5 printf("Le pid de mon fils est %d\n", pid);
        wait();
    }
    return 0;
}
```

Chapitre 2. Processus K. ElBedoui 87

PRIMITIVES DE GESTION DES PROCESSUS

2

```
1 Je suis le fils, mon pid est 5716
Le pid de mon père est 5715
corail 610
drwxr-xr-x 2 root root
drwxr-xr-x 2 root root
drwxr-xr-x 4 root root
drwxr-xr-x 2 root root
drwxr-xr-x 21 root root
drwxr-xr-x 75 root root
drwxr-xr-x 2 root root
drwxr-xr-x 108 root root
drwxr-xr-x 80 root root
drwxr-xr-x 2 root root
drwxr-xr-x 12 root root
drwxr-xr-x 2 root root
drwxr-xr-x 2 root root
drwxr-xr-x 4 root root
drwxr-xr-x 1 root root
drwxr-xr-x 2 root root
drwxr-xr-x 92 root root
drwxr-xr-x 34 root root
drwxr-xr-x 2 root root
drwxr-xr-x 6 root root
drwxr-xr-x 2 root root
drwxr-xr-x 4 root root
drwxr-xr-x 2 root root
drwxr-xr-x 53 root root
drwxr-xr-x 17 root root
drwxr-xr-x 27 root root
je suis le père, mon pid est 5715
Le pid de mon fils est 5716
```

Chapitre 2. Processus K. ElBedoui 88

PRIMITIVES DE GESTION DES PROCESSUS

2

```
1 Je suis le fils, mon pid est 5716
Le pid de mon père est 5715
corail 610
drwxr-xr-x 2 root root
drwxr-xr-x 4 root root
drwxr-xr-x 2 root root
drwxr-xr-x 21 root root
drwxr-xr-x 75 root root
drwxr-xr-x 2 root root
drwxr-xr-x 108 root root
drwxr-xr-x 80 root root
drwxr-xr-x 2 root root
drwxr-xr-x 12 root root
drwxr-xr-x 2 root root
drwxr-xr-x 2 root root
drwxr-xr-x 4 root root
drwxr-xr-x 1 root root
drwxr-xr-x 2 root root
drwxr-xr-x 92 root root
drwxr-xr-x 34 root root
drwxr-xr-x 2 root root
drwxr-xr-x 6 root root
drwxr-xr-x 2 root root
drwxr-xr-x 4 root root
drwxr-xr-x 2 root root
drwxr-xr-x 53 root root
drwxr-xr-x 17 root root
drwxr-xr-x 27 root root
je suis le père, mon pid est 5715
Le pid de mon fils est 5716
```

Chapitre 2. Processus K. ElBedoui 88

PRIMITIVES DE GESTION DES PROCESSUS

2 PRIMITIVES DE GESTION DES PROCESSUS

1 Je suis le fils, mon pid est 5716

2 Le pid de mon père est 5715

3

```
total 610
drwxr-xr-x 2 root root 4096 mai 3 2008 BIDAYA
drwxr-xr-x 2 root root 4096 nov 20 2006 bin
drwxr-xr-x 4 root root 1024 fév 18 11:54 boot
drwxr-xr-x 2 root root 4096 oct 6 17:34 CDROM
drwxr-xr-x 21 root root 118784 fév 20 19:36 dev
drwxr-xr-x 75 root root 8192 fév 20 19:36 etc
drwxr-xr-x 2 root root 4096 jun 13 2008 FLASH
drwxr-xr-x 108 root root 8192 jun 13 2008 GRAVURE
drwxr-xr-x 80 root root 376832 fév 18 22:20 home
drwxr-xr-x 1 root root 4096 nov 20 2006 lib
drwxr-xr-x 2 root root 4096 sep 6 2006 lib
drwxr-xr-x 4 root root 4096 nov 20 2006 mnt
drwxr-xr-x 1 root root 762 nov 14 18:18 named.conf
drwxr-xr-x 2 root root 4096 jan 25 2003 opt
dr-xr-xr-x 92 root root 0 fév 20 21:34 proc
drwxr-xr-x 34 root root 4096 fév 21 05:06 root
drwxr-xr-x 2 root root 8192 nov 14 11:22 sbin
drwxr-xr-x 6 root root 11182 jun 27 2008 SCRIPTS-UTILITAIRES
drwxr-xr-x 1 root root 4096 mai 3 2008 STARTUP
drwxr-xr-x 2 root root 4096 nov 20 2006 tftpboot
drwxr-xr-x 4 root root 4096 fév 21 05:04 tmp
drwxr-xr-x 53 root root 4096 mai 5 2008 usr
drwxr-xr-x 17 root root 4096 iun 8 2008 var
```

execvp("ls","ls","-l","/",NULL);

4 Je suis le père, mon pid est 5715

5 Le pid de mon fils est 5716

89

K. ElBedoui

Chptre 2. Processus

FIN

3. Exécution d'un processus

- Il y a de nombreux cas où on désire lancer une commande externe au programme, sans pour autant remplacer le processus en cours. On peut par exemple avoir une application principale qui lance des sous-programmes indépendants, ou désirer faire appel à une commande système.

- Pour cela, nous disposons de la fonction system() déclarée ainsi dans <stdlib.h> :

```
int system (const char * commande); exemple : system("ls");
```
- Cette fonction invoque le shell en lui transmettant la commande fournie, puis revient après la fin de l'exécution.

90

K. ElBedoui

Chptre 2. Processus