Chapitre 01 : Étude des Processus UNIX/Linux

Dr Mandicou BA
mandicou.ba@esp.sn
http://www.mandicouba.net

Diplôme D'Ingénieur de Conception (DIC, 2e année)
Master Professionnel (1e année)
Options : GLSI, SRT, SSI et IABD

ECOLE SUPERIEURE POLYTECHNIQUE

www.esp.sn

Plan du Chapitre

1. Généralités sur les Processus
   - Programmes et processus
   - Propriétés d'un processus sous Linux
   - Naissance de processus et threads
2. Programmation des processus
   - Création de processus
   - Terminaison de processus
   - Exécution de processus et arguments
   - Études des primitives *exec*

Sommaire

1 Généralités sur les Processus
- Programmes et processus
- Propriétés d'un processus sous Linux
- Naissance de processus et threads

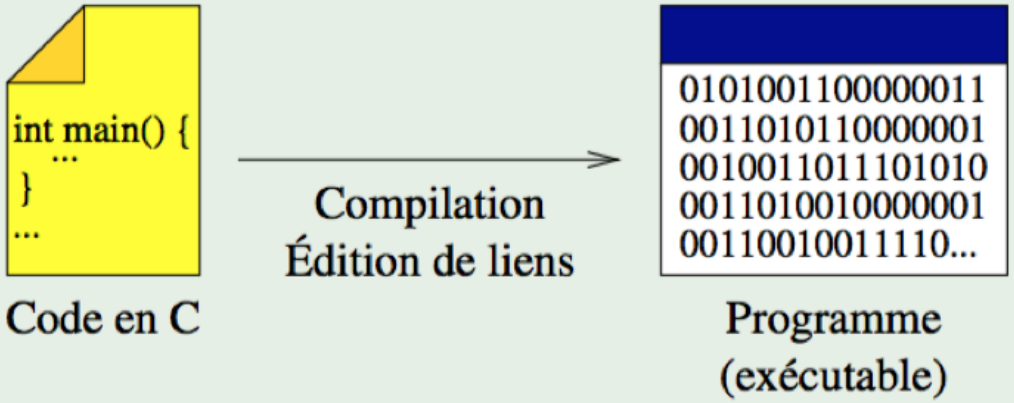
2 Programmation des processus

Programme

Définition

- Un programme est une suite d'instructions :
  - ↪ Code en C
  - ↪ Compilation
  - ↪ Édition de liens
- Chaque instruction est exécutée dans l'ordre par un processeur

Création d'un programme

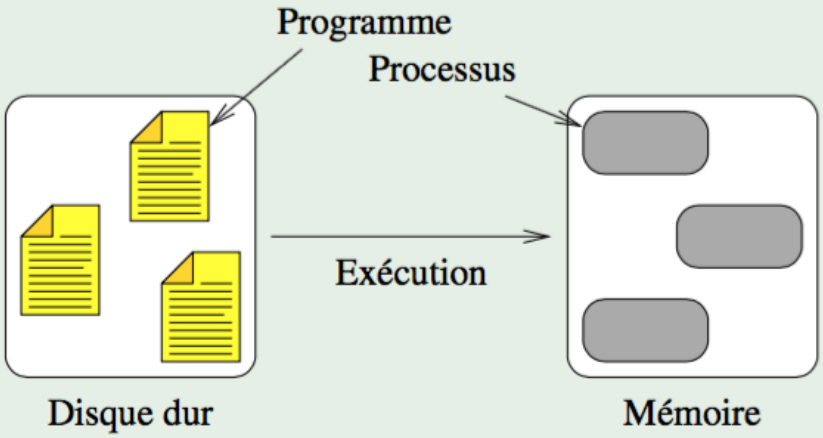


Le processus

Définition

- Programme placé en mémoire pour son exécution :
  - Instructions placées au fur-et-à-mesure dans le RI (Registre d'Instruction)
  - Numéro de la prochaine instruction dans le CO (Compteur Ordinal)
- Processus : programme en cours + états processeur et mémoire
  $\hookrightarrow$ Exécution au sein d'un espace d'adressage privé

Exécution d'un programme



Process Control Block

Description

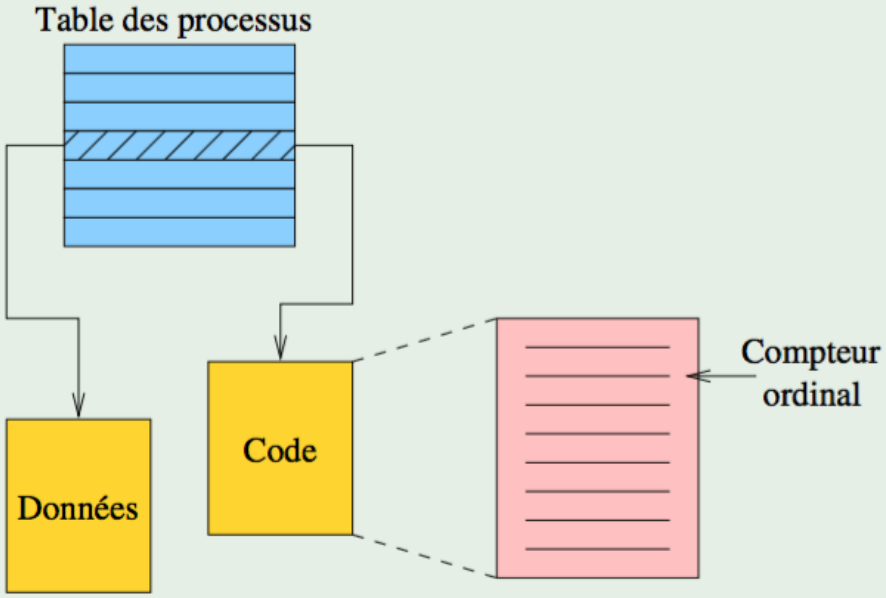
- Structure de description du processus (PCB) :
  ↪ Allouée dynamiquement par le système
- Contient :
  - Identifiant unique
  - État courant
  - Contexte processeur : compteur ordinal, registres processeur
  - Contexte mémoire : code, données
  - Information sur les ressources utilisées (fichiers)
  - *etc.*

La table des processus

Description

- L'ensemble des processus est géré dans une table :
  ↪ Une entrée par processus (PCB)

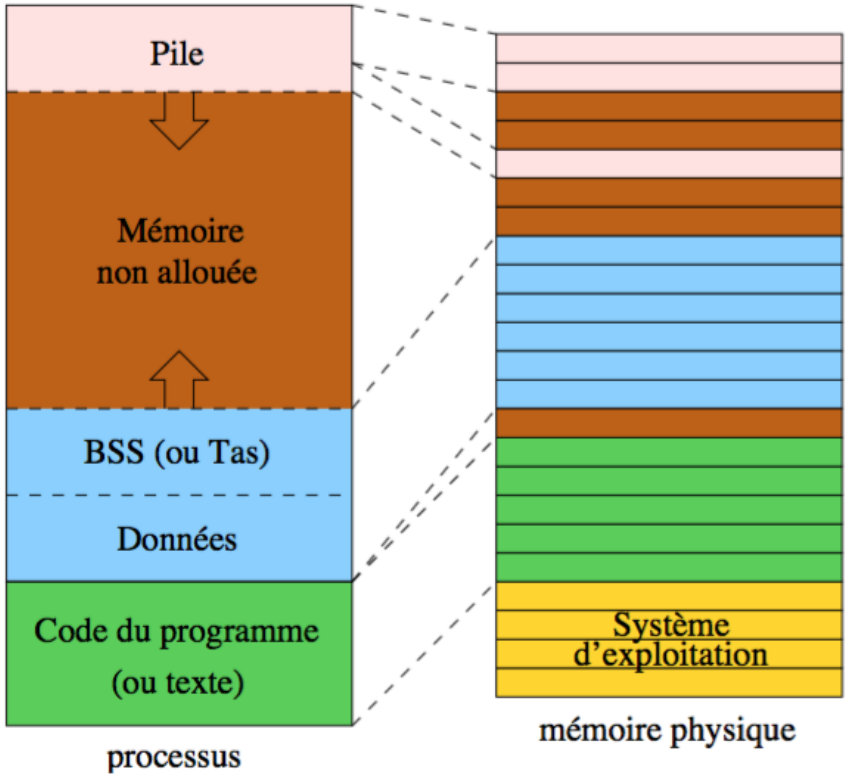
Représentation mémoire



Représentation mémoire d'un processus

Zones mémoire d'un processus

- Pile : appels de fonctions avec variables
- Données non initialisées (zone appelée BSS ou tas)
- Données initialisées et constantes
- Code du programme.



Contigüité des zones mémoire

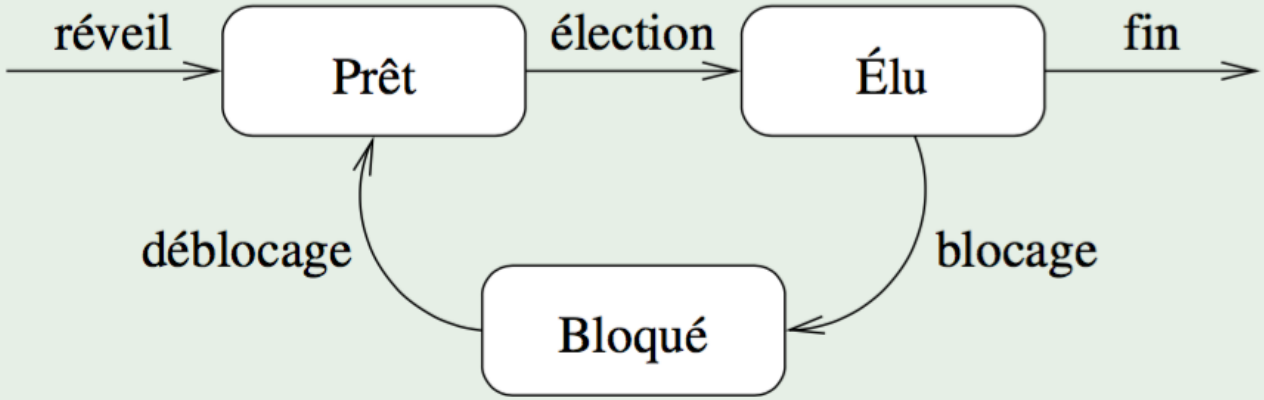
Les zones mémoire d'un processus ne sont pas contigües dans la mémoire physique.

États d'un processus

Description

- Chaque processus caractérisé par un état :
  ↪ Évolue au cours de son exécution
- Trois états différents :
  - Prêt : chargé en mémoire, en attente du processeur
  - Élu : élu par l'ordonnanceur, accès au processeur
  - Bloqué : attente d'une ressource

Diagramme d'états



Identifiant et filiation

L'identifiant

- À chaque processus est associé un identifiant :
  ↪ PID pour *Process IDentifier*
- Attribué par le système au moment de l'exécution
- Unique pour chaque processus
- Le premier processus : `init` possède le PID 1

Une filiation

- Un processus A créé par un processus B :
  ↪ A fils de B
  ↪ B père de A
- Le PPID d'un processus est le PID de son père

Le processus init (Linux)

Description

- Ancêtre de tous les processus *Linux* : processus 0
  ↪ Appelé par la fonction `start_kernel()`
- Occupe la première place de la table des processus :
  ↪ Initialisation des structures de données du noyau
  ↪ Autorise les interruptions
  ↪ Crée un second *thread* noyau : `init`
  ↪ Puis s'endort
- *Thread* `init` :
  ↪ Création de plusieurs *threads* de gestion (cache disque, swap)
  ↪ Exécution du programme `init` (primitive `execve`)
  ↪ Devient le processus `init`
- Tous les processus suivants sont créés à partir de `init` :
  ↪ Démons, processus utilisateurs

Filiation (suite)

Exemple de filiation (vue partielle de pstree)

```
rabat@ubuntu: ~
Fichier  Édition  Affichage  Terminal  Onglets  Aide
rabat@ubuntu:~$ pstree
init─┬─NetworkManager───{NetworkManager}
     ├─NetworkManagerD
     ├─gnome-terminal─┬─bash─┬─firefox───5*[{firefox}]
     │                │      └─pstree
     │                ├─gnome-pty-helpe
     │                └─{gnome-terminal}
     └─gnome-volume-ma
```

Cas de *Windows*

Le principe de filiation n'existe pas sous *Windows* :
tous les processus sont égaux.

Groupes et sessions de processus

Groupes de processus

- Processus organisés en groupes :
  ↪ Permet de faciliter la gestion
- Par défaut, un processus appartient au groupe de son père
- Chacun est identifié par un identifiant : PGID
  ↪ Le *leader* du groupe : PID = PGID
- Possibilité de créer un groupe :
  ↪ Utile pour gérer tous les processus fils en une seule fois

Sessions de processus

- Groupes regroupés en sessions identifiées par un SID :
  ↪ Normalement utiles que pour les terminaux ou les gestionnaires de fenêtres
- Par défaut, une session créée par terminal

Priorité et accès aux ressources

Priorité

- Utilisée lors de l'exécution du processus
- Un processus sera exécuté avant (ou plus souvent) que les autres processus moins prioritaires
- Généralement, la priorité évolue dans le temps

Accès aux ressources

- À chaque processus sont attribués des droits pour les accès aux ressources :
  ↪ Différents types d'accès possibles
- Généralement, droits hérités de ceux de l'utilisateur :
  - UID : l'identifiant de l'utilisateur
  - GID : l'identifiant de groupe de l'utilisateur

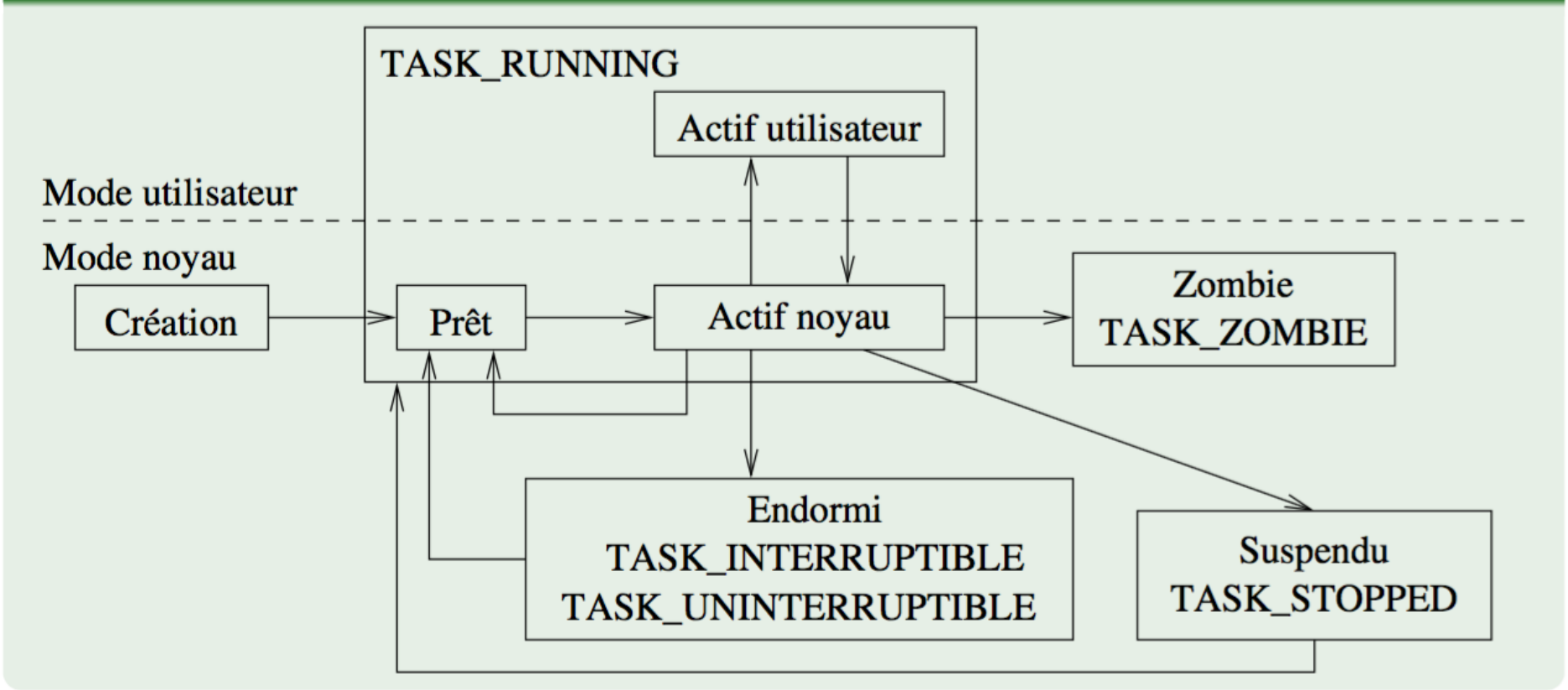
États des processus sous Linux (1/2)

Description

- Un processus peut être dans les états suivants :
  - `TASK_RUNNING` : prêt (état élu)
  - En sommeil (ou endormis) : bloqué
    - `TASK_UNINTERRUPTIBLE` : non interruptible (excepté par une interruption matérielle)
    - `TASK_INTERRUPTIBLE` : interruptible (par un signal)
  - `TASK_STOPPED` : suspendu (ou arrêté)
  - `TASK_ZOMBIE` : zombie

États des processus sous Linux (2/2)

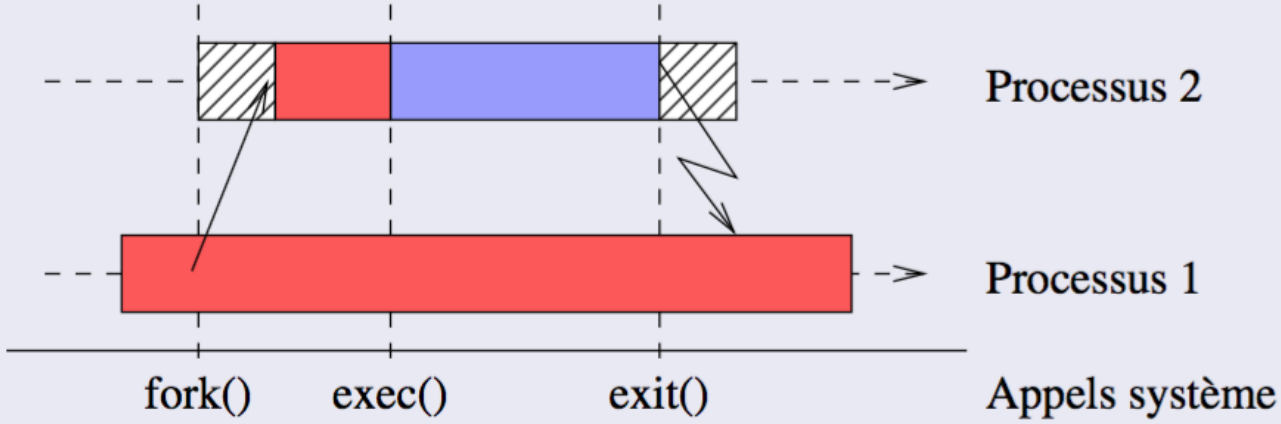
Cycle de vie d'un processus



Création de processus : cas de Linux (1/2)

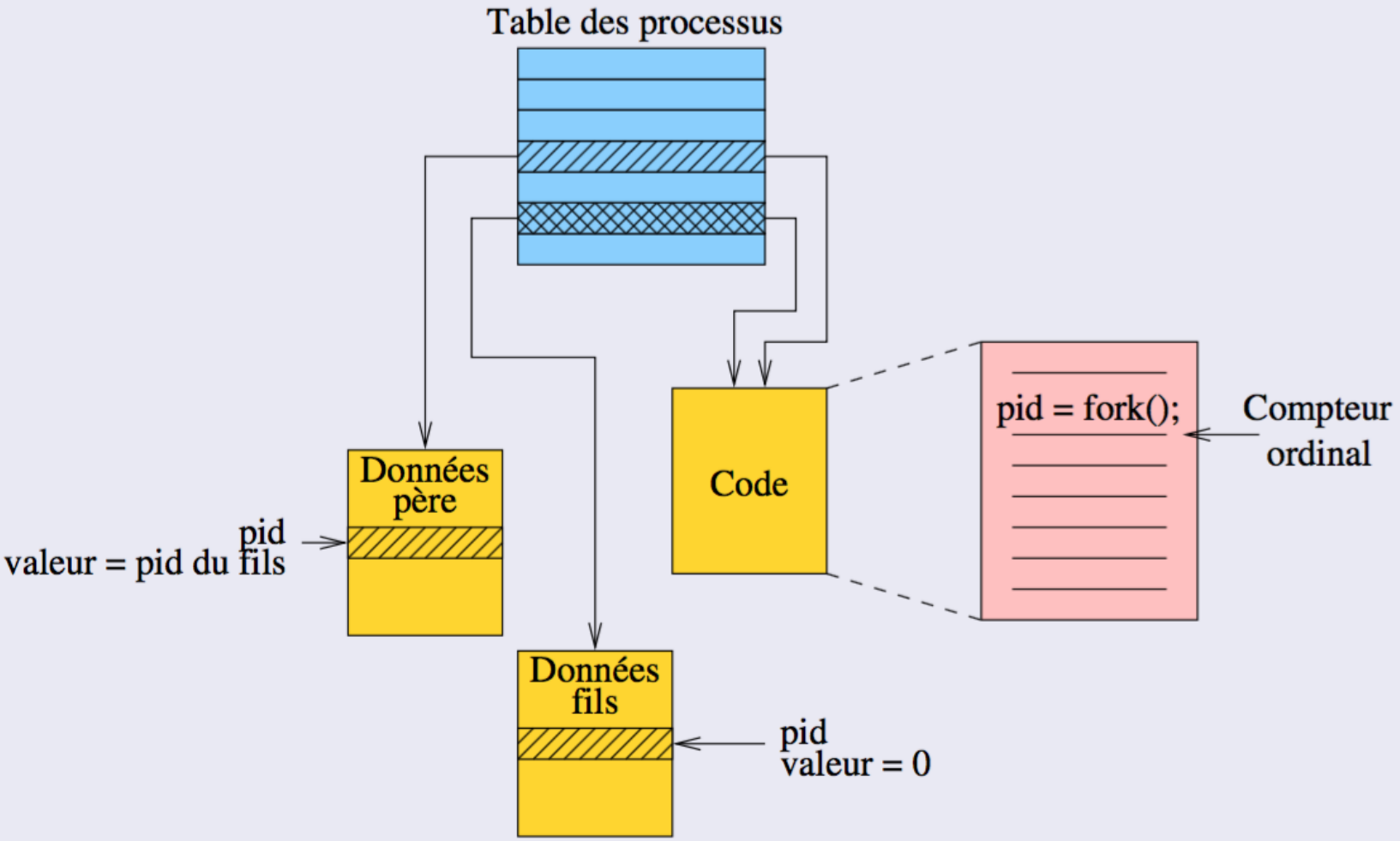
Sous *Linux*

- Appel système `fork()`
- Le processus appelant (le père) est dupliqué (même image mémoire)
- La copie est remplacée par le fils (via `exec`)



Création de processus : cas de Linux (2/2)

Création d'un processus avec `fork()`

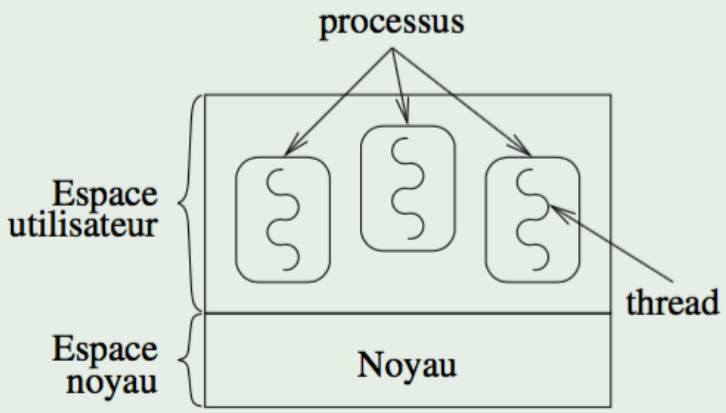


Les threads

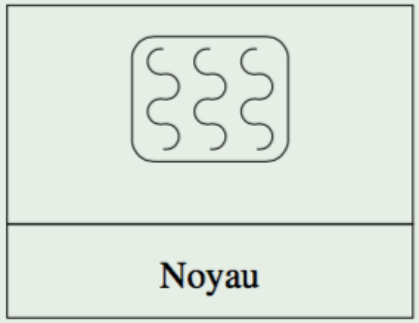
Définition

- Un processus peut être considéré par son *chemin d'exécution* :
  ↪ Appelé le *thread*
- Par défaut, un processus ne comporte qu'un seul *thread*
- Il est possible d'exécuter plusieurs *threads* par processus :
  - L'espace mémoire du processus est alors partagé entre les *threads*

Processus monothread et multithread



Processus monothread



Processus multithread

Exécution des threads

Description

- On appelle le multithreading, le fait que plusieurs *threads* soient présents dans le même processus
- L'exécution des *threads* est réalisée à tour de rôle...
  ↪ ...sur un mono-processeur !

L'espace d'adressage commun

- Les *threads* partagent tous le même espace d'adressage
- Les variables globales sont donc accessibles (et modifiables) par tous :
  ↪ Problèmes de concurrence
- Chaque *thread* possède sa propre pile d'exécution

Sommaire

1 Généralités sur les Processus

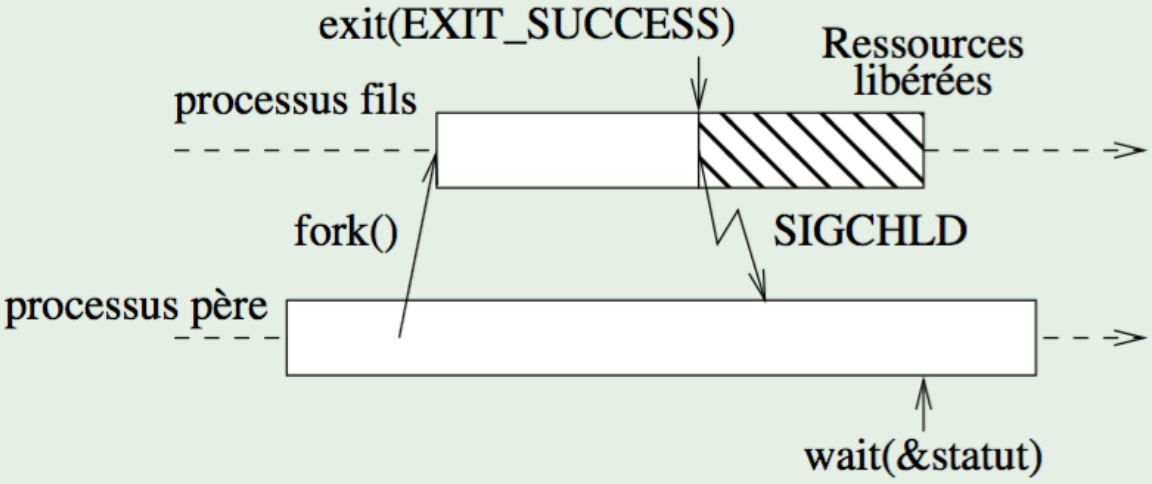
2 Programmation des processus
- Création de processus
- Terminaison de processus
- Exécution de processus et arguments
- Études des primitives *exec*

Description générale

Description

- Création d'un fils à l'aide de `fork`
- Fin du fils à l'aide de `exit` :
  ↪ Envoi d'un signal "`SIGCHLD`" au père + un statut
- Attente de la fin d'un fils avec `wait` ou `waitpid` :
  ↪ Libération des ressources du fils

Illustration



Création d'un processus fils

En-tête de la fonction (S2)

- `pid_t fork()`
- *Inclusion* : `unistd.h`

Valeurs retournées et erreurs générées

- Valeurs de retour :
  - < 0 : une erreur (pas de processus créé)
  - 0 : dans le processus fils
  - > 0 : dans le père, correspondant au PID du processus fils créé
- Erreurs possibles :
  - `EAGAIN` : trop de processus créés ou pas assez de mémoire pour allouer les structures
  - `ENOMEM` : pas assez de mémoire pour le noyau

Utilisation générale

Extrait de code

```
/* Avant */
pid_t pid;
pid = fork();
if (pid > 0) {
  /* Exécuté par le processus père */
} else if (pid == 0) {
  /* Exécuté par le processus fils */
} else {
  /* Traitement d'erreur */
}
/* Suite du code */
```

Remarques

- La suite du code est exécutée par les deux processus :
  ↪ À éviter en général (mettre un `exit` à la fin du code du fils)
- Pas de variables communes entre les deux processus

Exemple d'utilisation de fork() (1/2)

Programme d'exemple (sans inclusion, ni gestion d'erreur)

```
int global = 1;
int main(int argc, char *argv[]) {
  int i = 1;
  if(fork() == 0) {
    global++; i++;
  } else {
    global--; i--;
  }
  printf("PID=%d, global=%d, i=%d\n", getpid(), global, i);
  return EXIT_SUCCESS;
}
```

Question

- Qu'est-ce que ça affichera ? (à part les PID...)

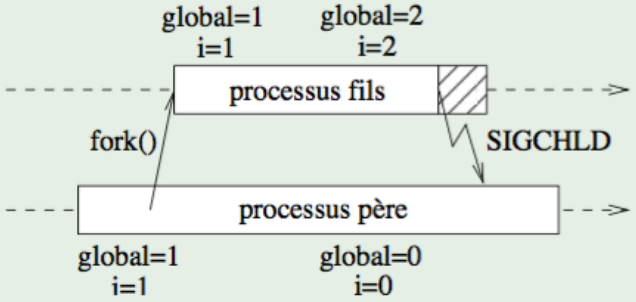
Exemple d'utilisation de fork() (2/2)

Programme d'exemple (sans inclusion, ni gestion d'erreur)

```
int global = 1;
int main(int argc, char *argv[]) {
  int i = 1;
  if(fork() == 0) {
    global++; i++;
  } else {
    global--; i--;
  }
  printf("PID=%d, global=%d, i=%d\n", getpid(), global, i);
  return EXIT_SUCCESS;
}
```

Exemple d'exécution

```
PID=3453, global=0, i=0
PID=3454, global=2, i=2
```



Identifiants de processus

Récupérer les identifiants d'un processus

- En-têtes des fonctions concernant les identifiants (S2) :
  - `pid_t getpid(void)` : retourne le PID du processus courant
  - `pid_t getppid(void)` : retourne le PID du processus père
  - `pid_t getpgid(pid_t pid)` : retourne l'ID du groupe du processus PID
  - `pid_t getsid(pid_t pid)` : retourne l'ID de session du processus PID
  - `pid_t getuid(void)` : retourne l'UID du processus
  - `pid_t getgid(void)` : retourne le GID du processus
  - *Inclusions* :
    - `unistd.h` pour les fonctions
    - `sys/types.h` pour les types

Attention

Attention aux exigences de macros suivant les fonctions !

Terminaison d'un processus

En-tête de la fonction (S3)

- `void exit(int statut)`
- *Inclusion* : `stdlib.h`

Explications

- Termine le processus en cours
- Les flux ouverts sont vidés et fermés

Paramètre

- `statut` : valeur & 255 retournée au processus père
- Constantes :
  ↪ `EXIT_SUCCESS` : terminaison normale
  ↪ `EXIT_FAILURE` : fin anormale

Attente de la fin d'un fils quelconque

En-tête de la fonction (S2)

- `pid_t wait(int *statut)`
- *Inclusions* : `sys/wait.h` pour la fonction et `sys/types.h` pour les types

Explications

- Suspend le processus en cours et attend la fin d'un des fils
- Si un fils est déjà terminé, pas de blocage
- Libère les ressources associées au fils

Paramètre

- `int *statut` : si différent de `null`, stocke la valeur de retour

Valeur retournée et erreurs générées

- Retourne le PID du fils terminé ou -1 en cas d'erreur
- Erreurs possibles :
  - `ECHILD` : pas de fils à attendre
  - `EINTR` : interruption par un signal

Macros pour l'analyse du statut

Description

- Utilisation générale : `NOM_MACRO(statut)`

Fin normale et valeur retournée

- `WIFEXITED` : vrai si le processus fils s'est terminé normalement
  ↪ Avec `exit` ou `return` depuis le `main`
- `WEXITSTATUS` : valeur retournée
  ↪ Attention : uniquement les 8 bits de poids faibles envoyés par `exit`

Fins anormales ou changements d'état

- `WIFSIGNALED` : vrai si fils terminé à la suite d'un signal non intercepté
- `WTERMSIG` : numéro du signal qui a terminé le fils (si `WIFSIGNALED` est vrai)
- `WIFSTOPPED` : vrai si le fils est arrêté (et option `WUNTRACED`)
- `WSTOPSIG` : numéro du signal qui a causé l'arrêt du fils (si `WIFSIGNALED` est non nul)

Exemple d'utilisation

Exemple de code (sans les inclusions)

```
void fils() {
  printf("Je suis dans le fils\n");
  exit(EXIT_SUCCESS);
}

int main() {
  pid_t pid;
  int statut;

  pid = fork();
  if(pid == -1) {
    perror("Erreur lors de la creation du processus fils ");
    exit(EXIT_FAILURE);
  }
  if(pid == 0)
    fils();
  printf("Je suis dans le pere et j'attends la fin du fils\n");
  if(wait(&statut) == -1) {
    perror("Erreur lors de l'attente du fils "); exit(EXIT_FAILURE);
  }
  if(WIFEXITED(statut))
    printf("Le fils a termine; valeur retournee = %d\n", WEXITSTATUS(statut));
  else
    printf("Le fils a termine anormalement.\n");
  return EXIT_SUCCESS;
}
```

Attente de la fin de fils spécifiques (1/2)

En-tête de la fonction (S2)

- `pid_t waitpid(pid_t pid, int *statut, int options)`
- *Inclusions* : `sys/wait.h` pour la fonction et `sys/types.h` pour les types

Explications

- Comme `wait` mais permet un contrôle plus fin

Paramètres

- `pid` : le PID du fils à attendre
  ↪ <-1 : fin d'un processus fils du groupe d'ID -PID
  ↪ -1 : fin d'un fils (comme `wait`)
  ↪ 0 : fin d'un fils du groupe de l'appelant
  ↪ >0 : fin du fils qui possède ce PID
- `options` (quelques valeurs) :
  ↪ `WNOHANG` : pas de blocage si aucun fils terminé
  ↪ `WUNTRACED` : indication sur les fils bloqués (non reçue précédemment)

Attente de la fin de fils spécifiques (2/2)

Valeur retournée et erreurs possibles

- PID du fils dont l'état à changé
- 0 si `WNOHANG` utilisé et qu'un fils dont le PID spécifié existe mais n'a pas changé d'état
- Erreurs possibles :
  - `ECHILD` : pas de fils à attendre
  - `EINTR` : interruption par un signal

Modification du code par rapport à `wait`

```
...
printf("Je suis dans le pere et j'attends la fin du fils\n");
if(waitpid(pid, &statut, 0) == -1) {
  perror("Erreur lors de l'attente du fils "); exit(EXIT_FAILURE);
}
if(WIFEXITED(statut))
  printf("Le fils a termine; valeur retournee = %d\n", WEXITSTATUS(statut));
else
  printf("Le fils a termine anormalement.\n");
...
```

Spécifier des procédures à appeler lors du exit

Description

- Possible d'enregistrer un ensemble de procédures appelées à la fin de l'exécution avec `atexit` :
  ↪ Pas de paramètre
- Les procédures sont empilées :
  ↪ Appelées dès l'appel à `exit` (ou `return` dans le `main`)
  ↪ Elles sont appelées dans l'ordre inverse de l'enregistrement
- Ne pas utiliser `exit` dans les procédures enregistrées :
  ↪ Comportement non standardisé
- Appel système `_exit` : aucune procédure appelée (ou les suivantes)

Attention

La fonction `on_exit` permet de spécifier en plus des procédures, des paramètres à passer mais elle n'est pas normalisée !

Enregistrement d'une procédure

En-tête de la fonction (S3)

- `int atexit(void (*procedure)(void))`
- *Inclusion* : `stdlib.h`

Paramètre

- `procedure` : pointeur sur la procédure à enregistrer

Valeur retournée

- Retourne 0 en cas de réussite, -1 dans le cas contraire

Arrêt immédiat d'un processus

En-tête de la fonction (S2)

- `void _exit(int code)`
- *Inclusion* : `unistd.h`

Explications

- Stoppe le processus immédiatement
- Ferme les descripteurs de fichier

Paramètre

- `code` : valeur retournée au processus père (comme pour `exit`)

Exemple d'utilisation (1/2)

Exemple de code (sans les inclusions)

```
void methode1() {
  printf("Je␣suis␣dans␣la␣methode␣1\n");
}

void methode2() {
  printf("Je␣suis␣dans␣la␣methode␣2\n");
}

int main() {
  if(atexit(methode1) != 0) {
    perror("Probleme␣lors␣de␣l'enregistrement␣");
    exit(EXIT_FAILURE);
  }

  if(atexit(methode2) != 0) {
    perror("Probleme␣lors␣de␣l'enregistrement␣");
    exit(EXIT_FAILURE);
  }

  printf("Ok,␣c'est␣termine\n");

  return EXIT_SUCCESS;
}
```

Exemple d'utilisation (2/2)

Affichage

```
Ok, c'est termine
Je suis dans la methode 2
Je suis dans la methode 1
```

Programme et arguments

Description

- Appel d'un programme : `ls -a -l`
  - `ls` : le nom du programme
  - "`-a`" et "`-l`" : les arguments
- Fonction principale en C : `int main(int argc, char *argv[])`
  Où :
  - `argc` : le nombre d'arguments
  - `argv` : tableau contenant les arguments

Exemple : `./toto 12 truc.txt`

- `argc` = 3
- `argv[0]` = "./toto"
- `argv[1]` = "12"
- `argv[2]` = "truc.txt"

Variables d'environnements

Description

- `int main(int argc, char *argv[], char **arge)`
- arge :
  ↪ Liste des variables d'environnement
  ↪ Chaque entrée, chaine contenant `NomVariable=Valeur`
- Exemple : `TMP=/tmp`, `PATH=...`

Parcours des variables

```
int main(int argc, char *argv[], char **arge) {
  int i = 0;
  while(arge[i] != NULL) {
    printf("%s\n", arge[i]);
    i++;
  }
  return EXIT_SUCCESS;
}
```

Manipuler les variables d'environnements (1/2)

Les différentes fonctions (S3)

- `char *getenv(const char *name)`
  ↪ Retourne la valeur associée à `name`
  ↪ NULL si pas de variable possédant ce nom
- `int putenv(const char *string)`
  ↪ Ajoute ou modifie une variable (dans ce cas, remplacement valeur)
  ↪ 0 en cas de réussite, -1 sinon (problème mémoire : `ENOMEM`)
  ↪ `string` de type `NOM=VALEUR`
- `int setenv(const char*name, const char *value, int overwrite)`
  ↪ Ajoute la variable si elle n'existe pas
  ↪ Si elle existe, la valeur est modifiée si `overwrite` est non nul
  ↪ 0 en cas de réussite, -1 sinon
- `void unsetenv(const char* name)`
- Autre solution : `externe **environ`
  ↪ Accès non protégé !

Chargement de processus : execve

Description

- En-tête de la fonction (S2) :

```
int execve(const char *fichier,
           char *const argv[],
           char *const envp[])
```

- *Inclusion* : `unistd.h`
- Remplace le processus courant par une instance du programme contenu dans `fichier`
- Il possible de spécifier des arguments : `argv`
  ↪ Tableau de chaînes dont la dernière est NULL
- Idem avec les variables d'environnement :
  ↪ `envp` peut être égal à NULL

Frontaux de execve

En-têtes des fonctions (S3)

- `int execl(const char *path, const char *arg0, ..., NULL)`
- `int execlp(const char *file, const char *arg0, ..., NULL)`
- `int execle(const char *path, const char *arg0, ..., char *const envp[])` :
- `int execv(const char *path, char *const argv[])`
- `int execvp(const char *file, char *const argv[])`
- *Inclusion* : `unistd.h`

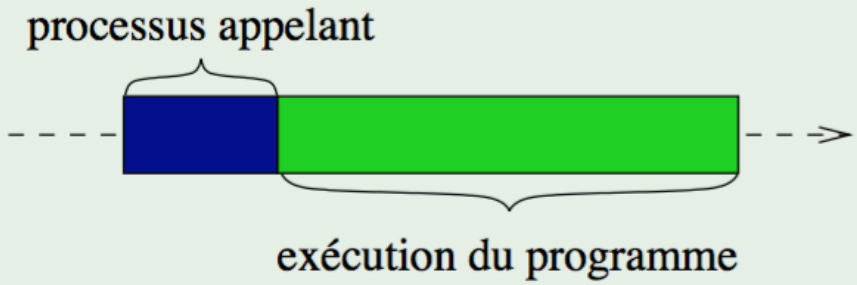
Conseil

Utilisation de execve conseillée !

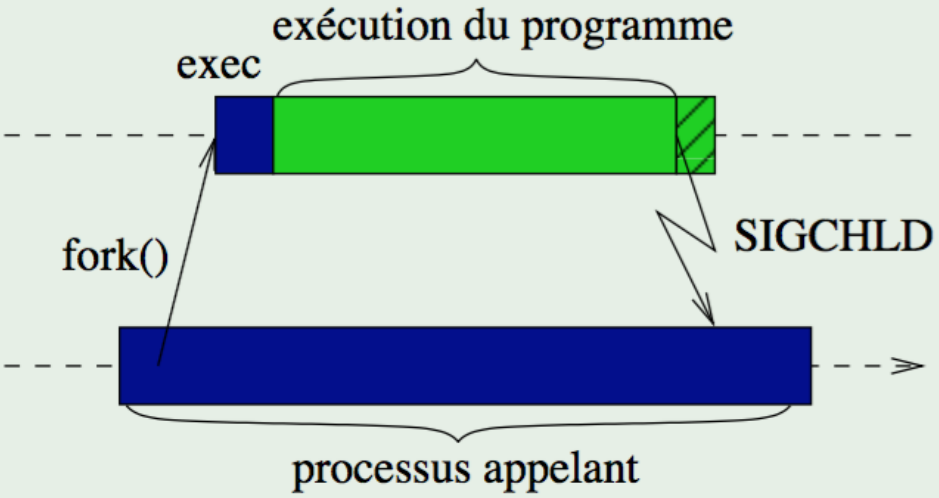
Principes de fonctionnement

Principe du exec

- Le processus courant est remplacé si l'appel est réussi.



Pour garder la main : `fork`

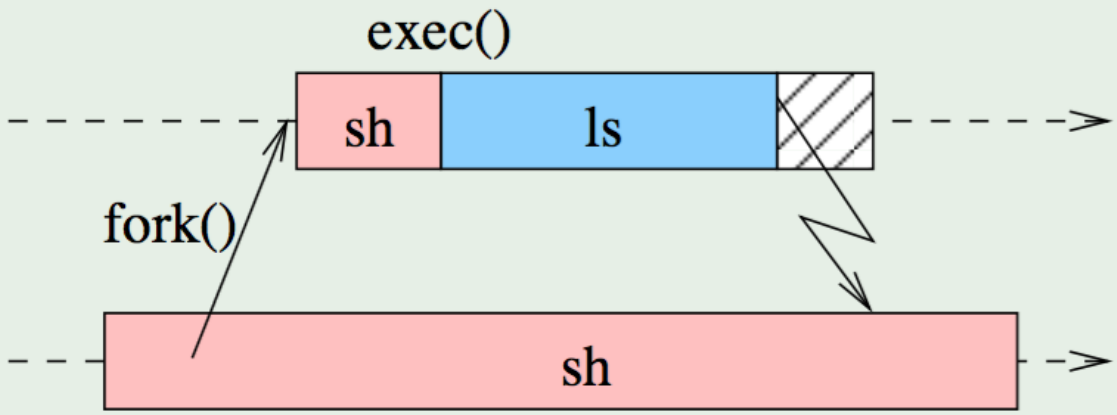


Exemple : appel d'une commande par le shell

Description

- Étapes de l'exécution d'une commande `ls` par le shell :
  1. Appel `fork()` : nouveau shell créé
  2. Appel `exec()` : exécution de la commande `ls`
  3. Le nouveau processus créé est remplacé par `ls`
  4. L'exécution est terminé et le processus se termine

Représentation



Et au niveau du processus ?

Description

- Les descripteurs de fichiers restent ouverts !
  ↪ Sauf configuration spécifique
- Au niveau des signaux :
  ↪ Ceux ignorés, restent ignorés
  ↪ Ceux dont un gestionnaire est spécifié : retour à défaut
- Modifications possibles des UID, GID
- Quelques autres attributs conservés :
  ↪ PID, PPID, répertoire de travail...

Exemple d'utilisation (1/3)

Code pour le programme `helloworld` (sans inclusion)

```
int main(int argc, char *argv[], char *argve[]) {
  int i;

  printf("***************\nHello World!!!\n");

  printf("\nLes arguments :\n");
  for(i = 0; i < argc; i++) {
    printf("%d : %s\n", i, argv[i]);
  }

  printf("\nLes variables d'environnement :\n");
  i = 0;
  while(argve[i] != NULL) {
    printf("%d : %s\n", i, argve[i]);
    i++;
  }
  if(i == 0)
    printf("Aucune\n");
  printf("***************\n");

  return EXIT_SUCCESS;
}
```

Exemple d'utilisation (2/3)

Code pour le programme execution (sans inclusion)

```
void execution(char *argve[]);

int main(int argc, char *argv[], char *argve[]) {
  pid_t pid;

  if((pid = fork()) == -1) {
    perror("Erreur lors de la creation du fils ");
    exit(EXIT_FAILURE);
  }
  if(pid == 0)
    execution(argve);

  if(waitpid(pid, NULL, 0) == -1) {
    perror("Erreur lors de l'attente de la fin du processus ");
    exit(EXIT_FAILURE);
  }

  return EXIT_SUCCESS;
}
...
```

Exemple d'utilisation (3/3)

Code pour le programme execution (sans inclusion) (suite)

```
...
void execution(char *argve[]) {
  char *arguments[6] = { "./helloworld", "Bonjour", "tout", "le", "monde",
       NULL };

  printf("Je demarre le programme...\n");
  if(execve("./helloworld", arguments, argve) == -1) {
    perror("Erreur lors du chargement ");
  }
  exit(EXIT_FAILURE);
}
```

Chapitre 01 : Étude des Processus UNIX/Linux

Dr Mandicou BA
mandicou.ba@esp.sn
http://www.mandicouba.net

Diplôme D'Ingénieur de Conception (DIC, 2e année)
Master Professionnel (1e année)
Options : GLSI, SRT, SSI et IABD

ECOLE SUPERIEURE POLYTECHNIQUE

www.esp.sn