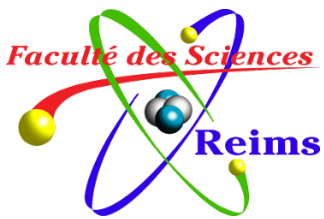
Les processus

Cyril Rabat
cyril.rabat@univ-reims.fr

Licence 3 Informatique - Info0601 - Systèmes d'exploitation - concepts avancés

2019-2020



Cours n°5

Représentation des processus

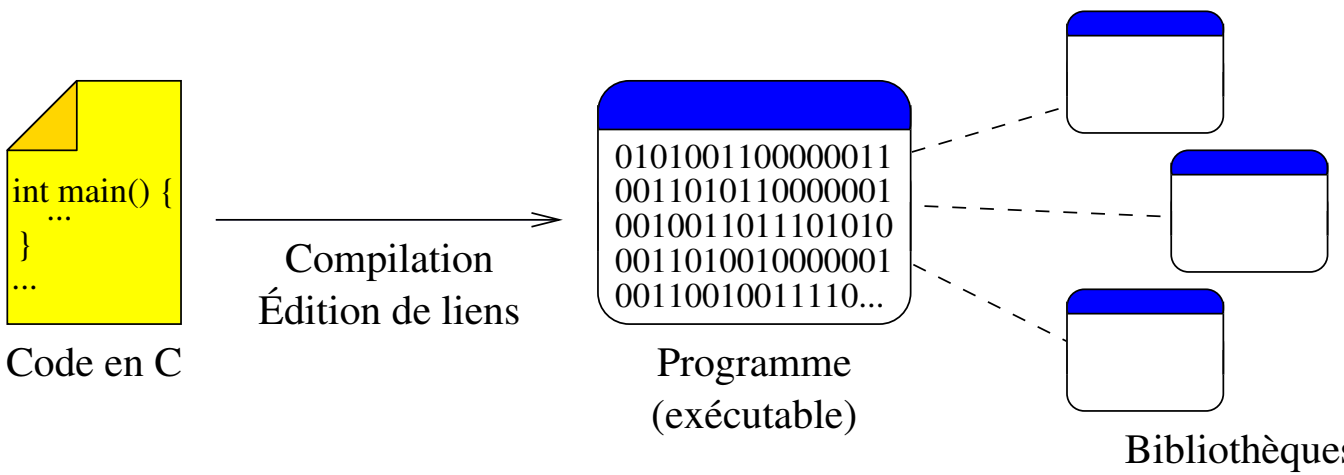
Gestion des processus (depuis le shell et en C)

Version 16 janvier 2020

Table des matières

Programme

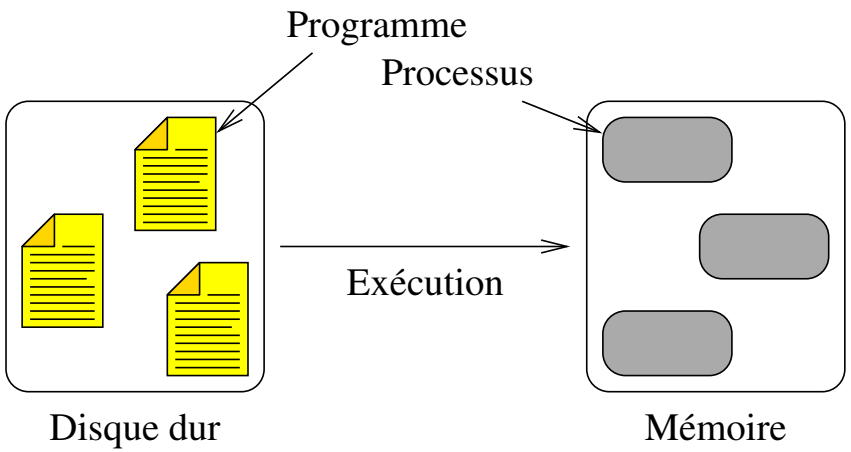
- Un programme est une suite d'instructions :
  ↪ Code en C (ou autre)
  ↪ Compilation
  ↪ Édition de liens
- Chaque instruction est exécutée dans l'ordre par un processeur



Création d'un programme

Le processus

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



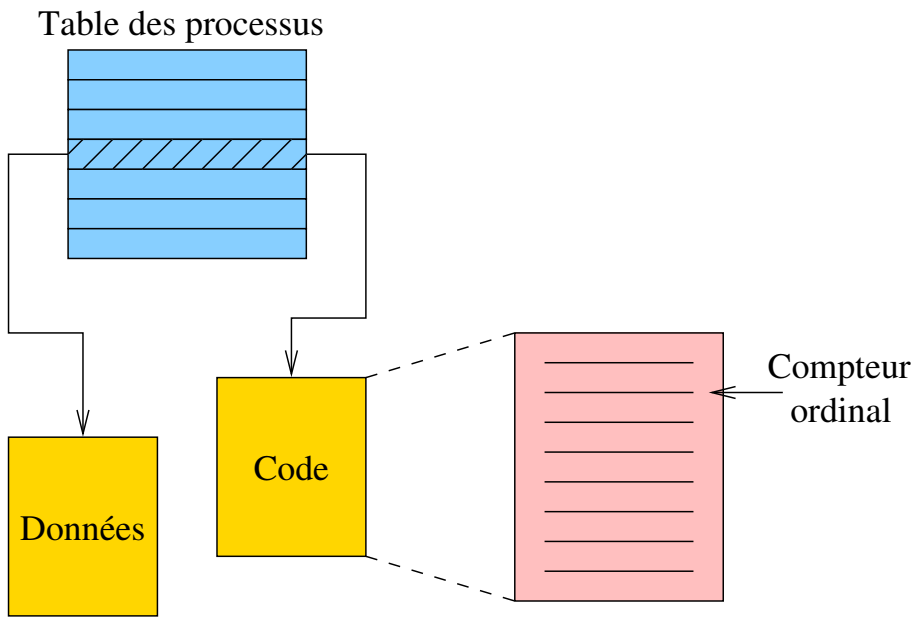
Exécution d'un programme

Process Control Block

- Structure de description du processus (PCB) :
  $\hookrightarrow$ 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

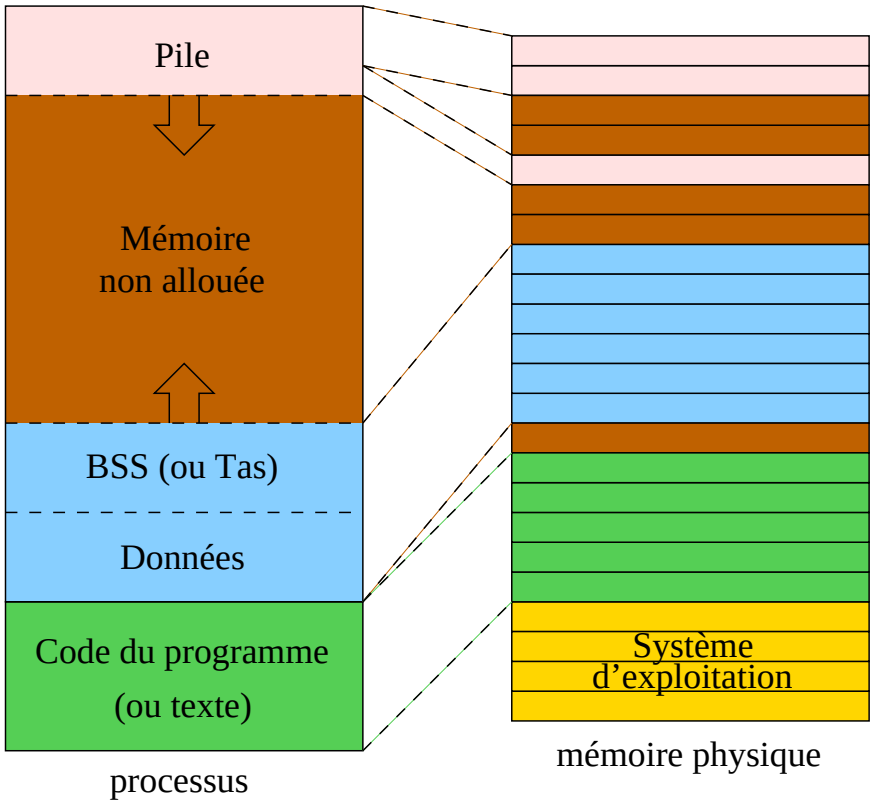
- L'ensemble des processus est géré dans une table :
  ↪ Une entrée par processus (PCB)



Représentation mémoire

Représentation 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.



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

États d'un processus

- Chaque processus caractérisé par un état :
  $\hookrightarrow$ 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

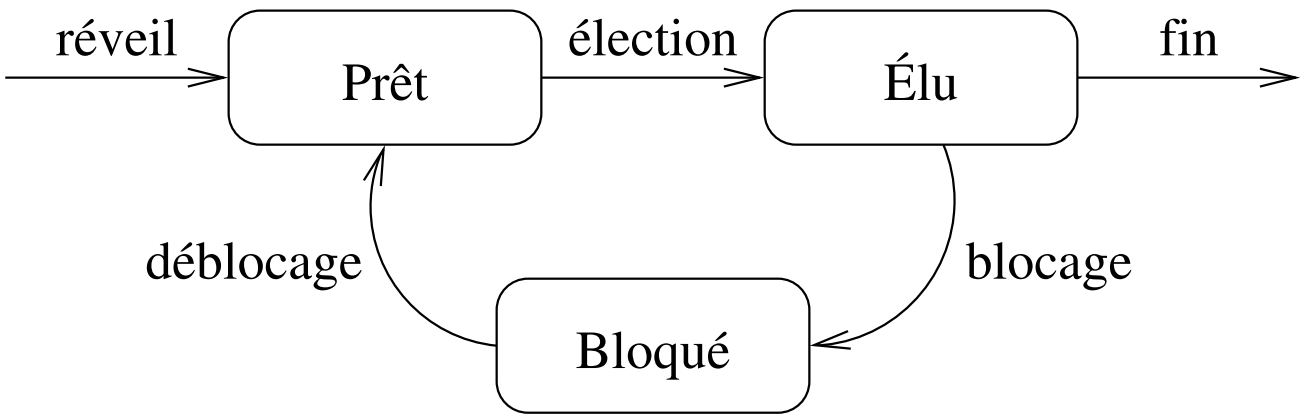


Diagramme d'états

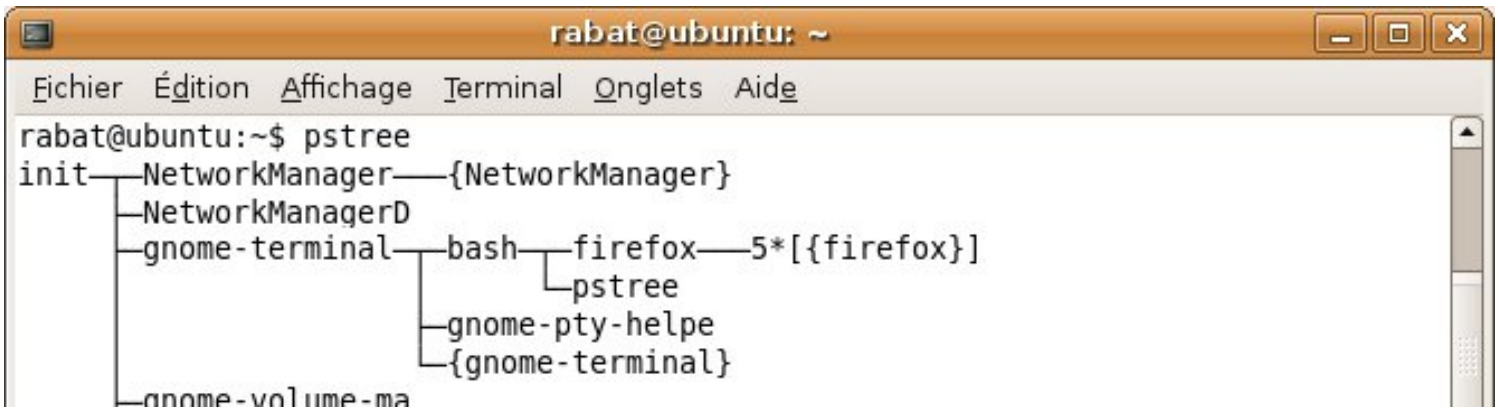
Identifiant et filiation

- 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
- 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*)

- 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 créés à partir de `init` :
  ↪ Démons, processus utilisateurs

Filiation (suite)



Exemple de filiation (vue partielle de `pstree`*)*

Cas de *Windows*

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

Groupes et sessions 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
- 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 :
    $\hookrightarrow$ 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

Rappels sur les UID et GID

- *Unix/Linux* : systèmes orientés multi-utilisateurs
- Utilisateurs regroupés en groupes d'utilisateurs :
  ↪ Permet de définir un ensemble de droits/restrictions
- Un utilisateur :
  ↪ Correspond à un nom
  ↪ Mais est associé à un identifiant noté UID (pour *User IDentifier*)
- Un processus appartient à l'utilisateur qui l'a créé et à son groupe
- Mais peut être modifié ! Du coup, trois UID et trois GID :
  ↪ Réel, effectif, sauvé

Les groupes d'utilisateurs

- Définis dans le fichier `/etc/group`
- Affiche aussi l'appartenance aux groupes
- Par exemple, sous *Ubuntu* :
  - Pas possible de se connecter en administrateur (comme pour *Windows*)
  - Utilisation de la commande `sudo` pour avoir les privilèges administrateurs
  - Mais l'utilisateur doit appartenir au groupe `sudo`

Exemple : installation d'un package sous *Ubuntu*

```
sudo apt-get install emacs
```

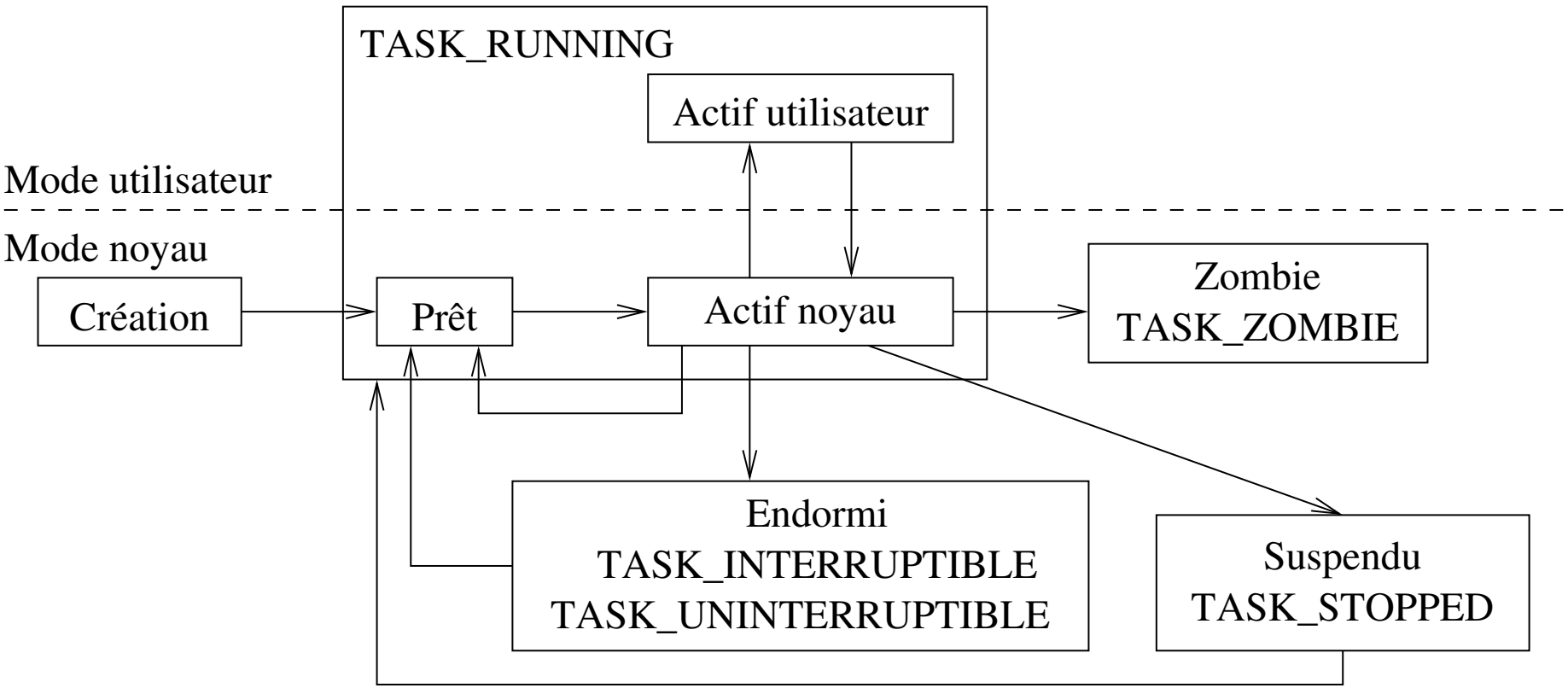
Remarque

Contrairement à *Ubuntu*, l'utilisateur n'est pas automatiquement dans le groupe `sudo` dans les distributions *Linux*.

États des processus sous *Linux* (1/2)

- `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

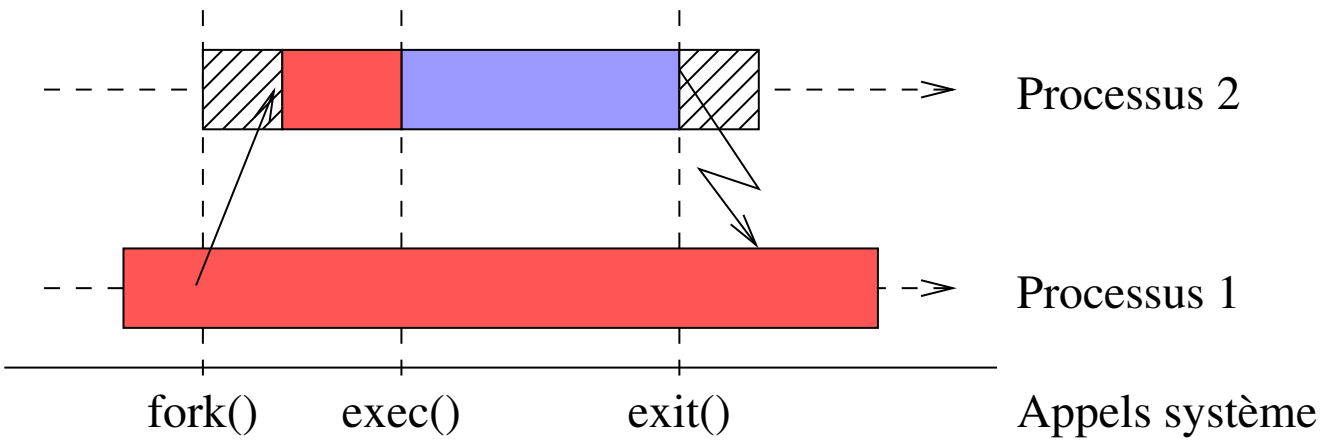
États des processus sous *Linux* (2/2)



Cycle de vie d'un processus

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

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



Exemple de création d'un processus

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

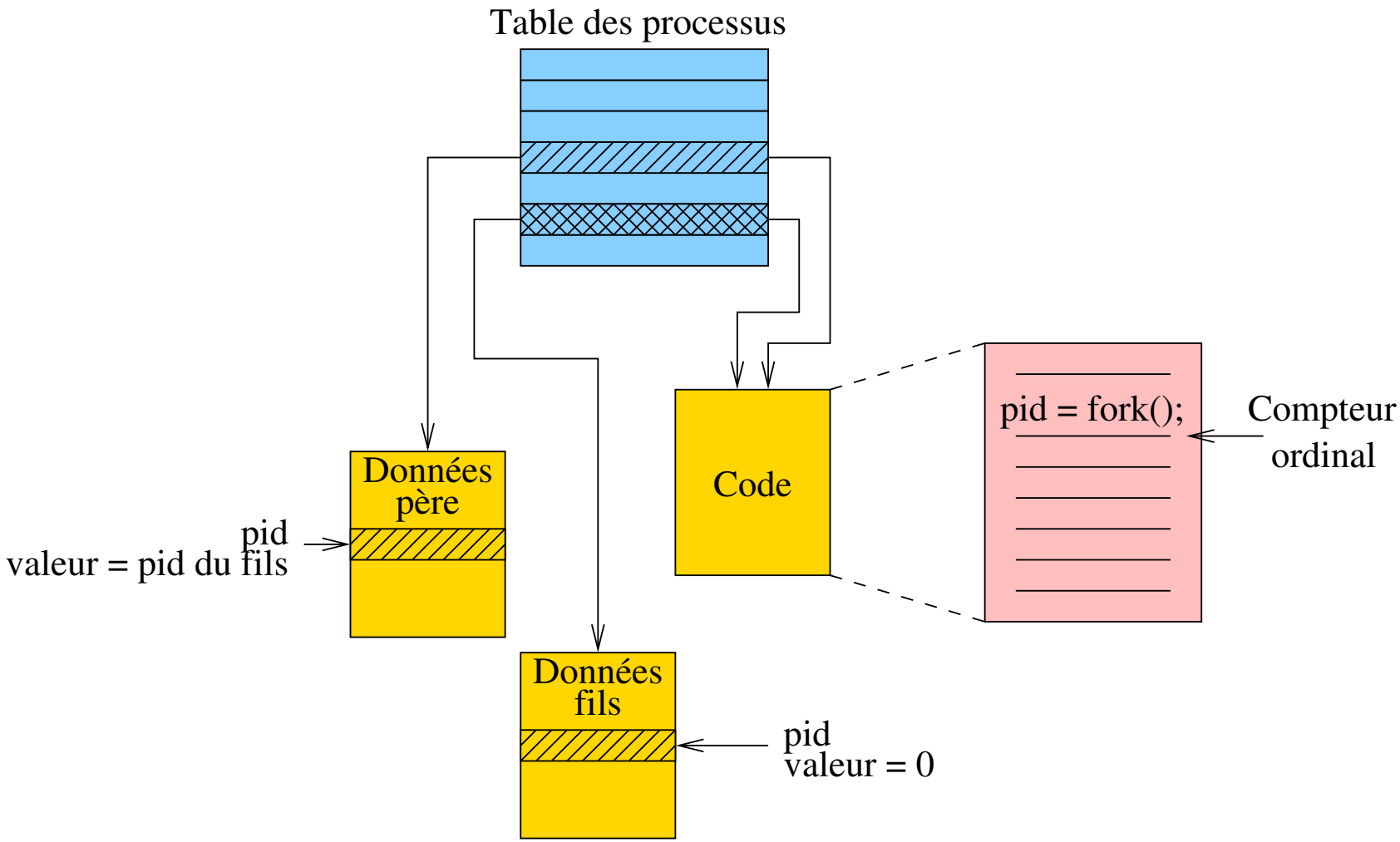
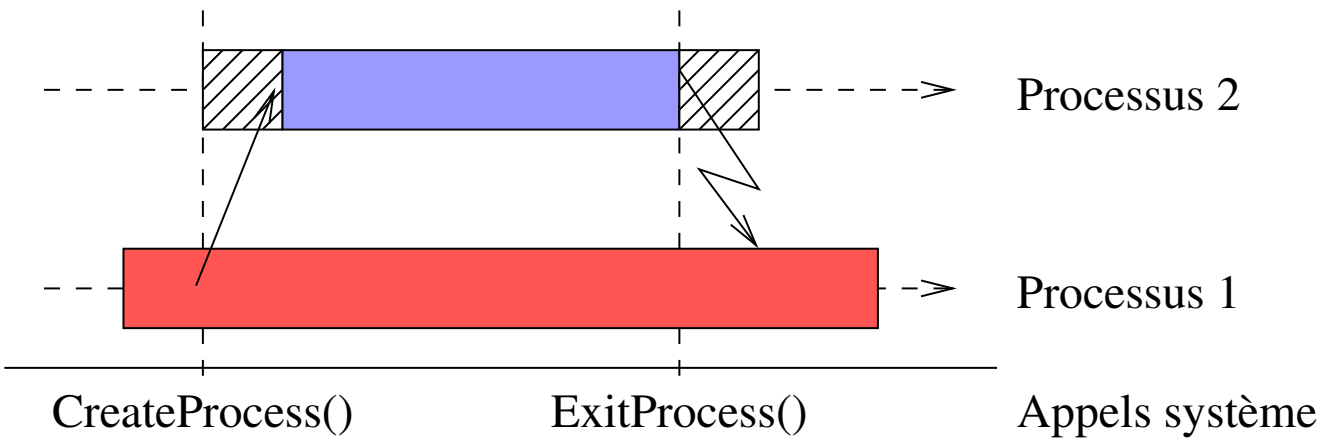


Illustration de l'effet de la création d'un processus avec `fork()`

Création de processus : cas de *Windows*

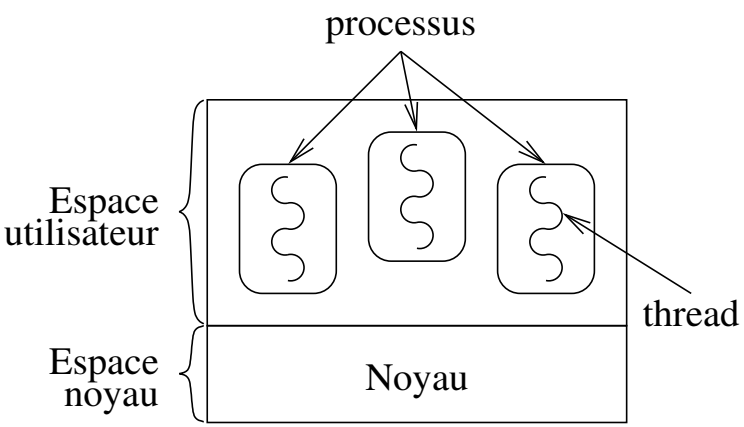
- Appel à `CreateProcess()` (*Win32*)
- Le processus fils est créé et remplacé directement par le processus



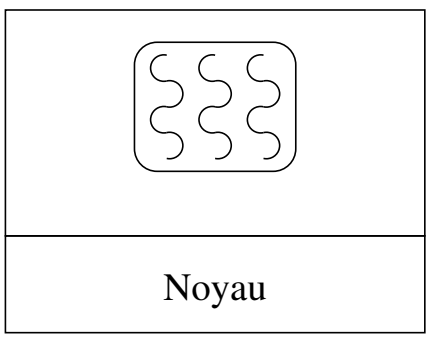
Exemple de création d'un processus

Les *threads*

- 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

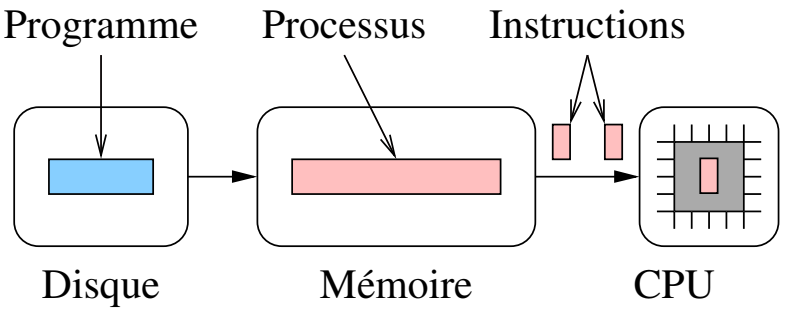


Processus multithread

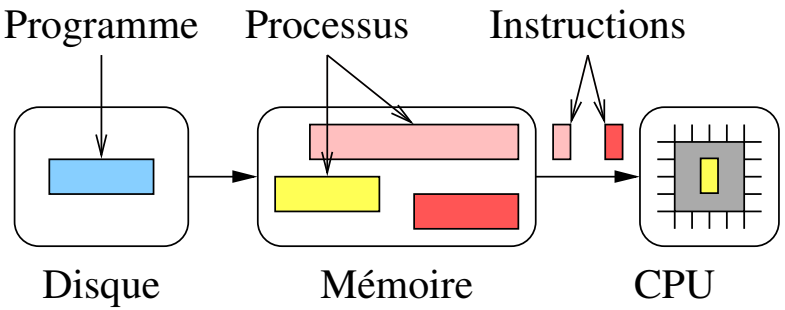
Exécution des *threads*

- 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 !
- 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

Problématique de l'ordonnancement



Monoprogrammation



Multiprogrammation

- Système monoprogrammé versus multiprogrammé
- Cas d'un système multiprogrammé :
  - Plusieurs processus en mémoire, un seul à la fois accède au CPU
  - Comment choisir le processus qui a accès au CPU ?
  - Comment changer cet accès au cours de l'exécution ?

Ordonnancement de processus

Définition : Ordonnanceur

- Basé sur un algorithme qui élit un processus :
  ↪ Ce processus a le privilège d'accéder au CPU

- Critères pour un bon ordonnancement :
  - Pourcentage d'utilisation CPU
  - Nombre de processus exécutés en un temps donné
  - Temps pour exécuter un processus
  - Temps d'attente d'un processus dans la file

Système interactif = temps de réponse court pour répondre à l'utilisateur

Commutation de contexte

Définition : Commutation de contexte

Une commutation de contexte consiste à sauvegarder l'état d'un processus et à restaurer l'état d'un autre processus.

- Ordonnanceur non préemptif :
  $\hookrightarrow$ Sélection d'un processus qui s'exécute jusqu'à ce qu'il libère volontairement le processeur
- Ordonnanceur préemptif :
  - Sélection d'un processus qui s'exécute pendant un délai déterminé
  - Si le processus est toujours en cours d'exécution après ce délai, il est suspendu et un autre processus est choisi

Ordonnancement sous *Linux* : généralités (1/2)

- Basé sur les *threads* (et non les processus)
- Trois classes de *threads* sont définies pour l'ordonnancement :
  - FIFO temps réel
  - *Round Robin* temps réel
  - *Threads* en temps partagé
- *Thread* FIFO temps réel :
  - Non préemtibles
  - Prioritaires par rapport aux autres

Ordonnancement sous *Linux* : généralités (2/2)

- *Thread Round Robin* temps réel :
  - Idem que la première catégorie excepté que les *threads* sont interruptibles par l'horloge
  - Si plusieurs *threads* éligibles : exécution selon l'algorithme *Round Robin*
- *Thread* en temps partagé :
  - *Threads* non prioritaires

Priorité et quantum

- Priorité d'ordonnancement entre 0 et 40
- Peut être modifiée par l'appel système `nice(val)`
- Plus la priorité est grande, plus le *thread* est rapide à répondre et reçoit une proportion de CPU plus importante
- À chaque *thread* est associé un quantum de temps :
  $\hookrightarrow$ Nombre de tops d'horloge
- L'ordonnanceur calcule une valeur de *bonté* d'un *thread* :
  - Si (classe = temps_réel) alors bonté $\leftarrow$ 1000 + priorité
  - Si (classe = temps_partagé et quantum $>$ 0) alors bonté $\leftarrow$ quantum + priorité
  - Si (classe = temps_partagé et quantum = 0) alors bonté $\leftarrow$ 0

L'ordonnancement

- Sélection du *thread* possédant la plus forte valeur *bonté*
- Pendant son exécution, son quantum est décrémenté de 1
- *Thread* retiré du CPU si :
  - Son quantum est à 0
  - Il est bloqué par un appel d'entrée/sortie, un sémaphore ou autre
  - Un *thread* de *bonté* supérieure jusque-là bloqué devient éligible
- Au fur-et-à-mesure de l'exécution des *threads* :
  $\hookrightarrow$ Leur quantum diminue jusqu'à 0
- Les *threads* bloqués : quantum différent de 0
- Une fois qu'aucun *thread* n'est plus éligible, mise à jour des quanta de tous les processus :

```
quantum = (quantum / 2) + priority
```

Conséquences de l'algorithme

- Un *thread* utilisant beaucoup le CPU :
  - Son quantum diminue rapidement
  - La nouvelle valeur sera égale à la priorité
- Un *thread* effectuant beaucoup d'entrées/sorties :
  - Son quantum ne sera pas égal à 0
  - La valeur de son quantum va augmenter (jusqu'à sa priorité $\times$ 2)
- Si plusieurs *threads* utilisant de la CPU sont en concurrence :
  - Leur quantum diminue vers priorité
  - Les *thread* les plus prioritaires obtiennent une portion plus importante du CPU

Le shell

- Sous *Linux* :
  - Appelé aussi interface en ligne de commande
  - Permet d'exécuter des commandes sous forme d'entrées textes
  - Plusieurs versions :
    ↪ `sh`, `csh`, `tcsh` et le plus répandu `bash`
- Sous *Windows* :
  - Appelé *Invite de commandes* :
    ↪ Interpréteur de commandes MS-DOS
  - Permet d'émuler MS-DOS avec possibilité d'exécuter des programmes
  - Commandes presque similaires au `bash`
  - Pour aller plus loin : *Power Shell*

Syntaxe des commandes en `bash`

- Par défaut, le prompt est affiché :
  ↪ Dépend de la configuration du `bash`
- Une commande est tapée sous la forme d'une chaine de caractères
- Lors de l'analyse de la chaine, elle est découpée en fonction des espaces :
  - La première partie correspondant au nom du programme à exécuter
  - Les autres parties sont les options passées au programme exécuté

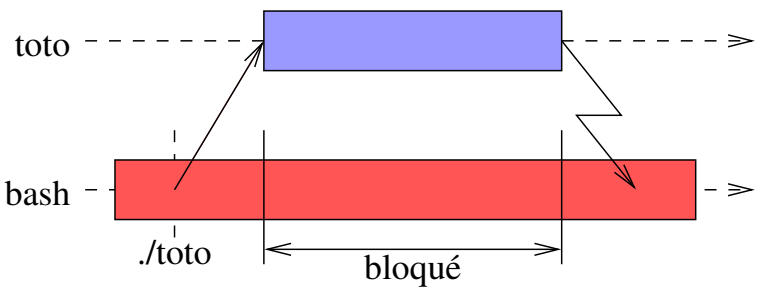
Exemple

```
rabat@ubuntu:~$ ls -a -l
```

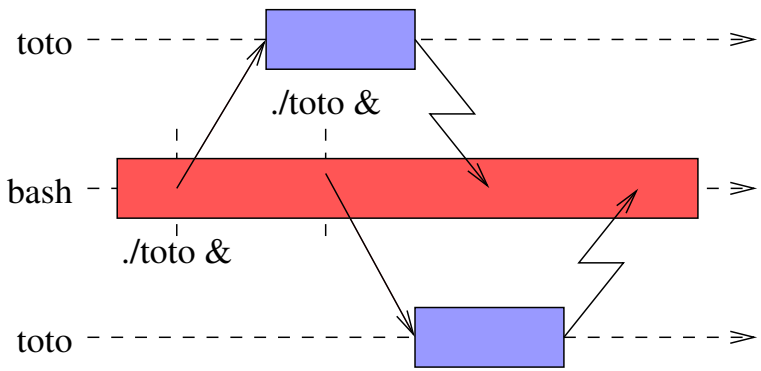
- `rabat@ubuntu:~$` : le prompt
- `ls` : la commande `ls` (liste des fichiers du répertoire courant)
- `-a` et `-l` : options de `ls`

Exécution de processus sous le `bash`

- En tapant directement le nom du programme
- Lors de l'exécution, l'interpréteur est *bloqué* :
  ↪ Aucune autre commande ne peut être exécutée
- Possibilité d'exécuter un programme en tâche de fond :
  ↪ `toto &`



`./toto`



`./toto &`

Exemple d'exécution d'un programme `toto`

Exécution en tâche de fond

Exemple d'exécution

```
rabat@ubuntu:~$ firefox &
[1] 6623
rabat@ubuntu:~$
```

- Le programme `firefox` est exécuté :
  ↪ Une nouvelle fenêtre s'ouvre contenant `firefox`
- Affichage du PID du processus correspondant à `firefox` (ici 6623)
- L'utilisateur récupère immédiatement la main

Fin d'exécution (en fermant la fenêtre de `firefox`)

```
rabat@ubuntu:~$
[1]+ Done                    firefox
rabat@ubuntu:~$
```

La commande `ps`

- *Rappel* : permet d'afficher la liste des processus en cours d'exécution
- Par défaut, uniquement les processus du même utilisateur et associés au terminal courant sont affichés
- Différentes options :
  - `ps -A` : affiche tous les processus
  - `ps -u toto` : affiche tous les processus de l'utilisateur `toto`

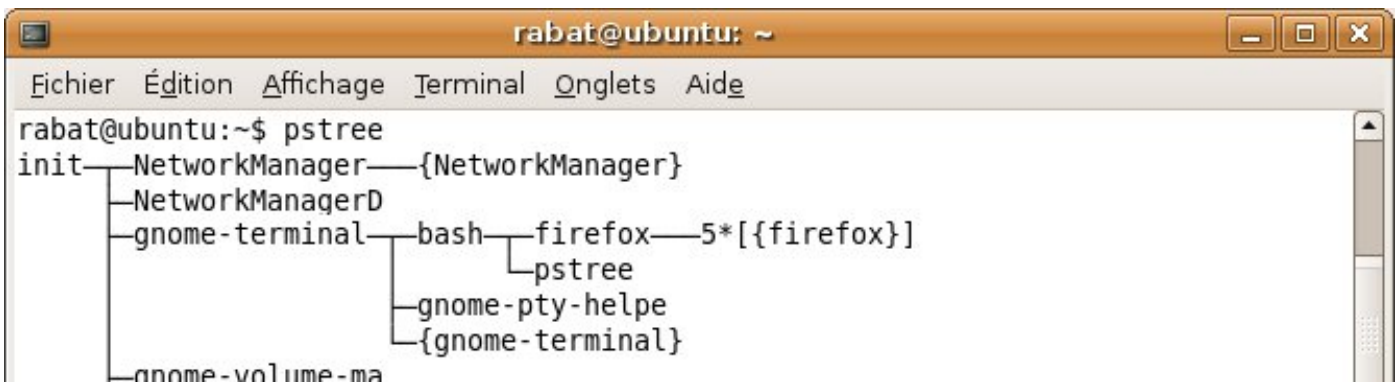
Exemple de `ps`

- Un appel simple à `ps` depuis un terminal

```
rabat@ubuntu:~> ps
  PID TTY          TIME CMD
 6681 pts/0    00:00:00 bash
 7049 pts/0    00:00:00 ps
```

La commande `pstree`

- *Rappel* : permet d'afficher l'arborescence des processus en cours d'exécution
- Par défaut, l'arborescence de tous les processus est affichée
- Quelques options :
  - `pstree -a` : affiche la ligne de commande associée au processus
  - `pstree -h` : affiche en gras le processus courant (`pstree`)
  - `pstree -p` : affiche les PID des processus



Exemple (partiel) de `pstree`

Observer les filiations avec `ps`

Exemple (partiel) de `ps xaf`

```
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     0:00 /sbin/init
...
 1988 ?        Sl     0:03 gnome-terminal
 1998 ?        S      0:00  \_ gnome-pty-helper
 1999 pts/2    Ss     0:00  \_ bash
 2566 pts/2    S+     0:00  |   \_ ./toto
 2427 pts/3    Ss     0:00  \_ bash
 2570 pts/3    R+     0:00      \_ ps xaf
...
```

- TTY : terminal associé au processus
- STAT : état du processus
  ↪ Premier caractère : S pour *sleep*, R pour *runnable*...
  ↪ Deuxième caractère : s pour *session leader*, + pour *foreground process*, etc.

Affichage des différents identifiants

Exemple (partiel) de `ps axj`

```
 PPID   PID  PGID   SID TTY   TPGID STAT   UID  TIME COMMAND
    0     1     1     1 ?        -1 Ss       0  0:00 /sbin/init
...
    1  1988  1581  1581 ?        -1 Sl    1000  0:11 gnome-term
 1988  1999  1999  1999 pts/2  3476 Ss    1000  0:00 bash
...
 1988  2427  2427  2427 pts/3  3479 Ss    1000  0:00 bash
...
 1999  3476  3476  1999 pts/2  3476 S+    1000  0:00 ./toto1
 3476  3477  3476  1999 pts/2  3476 S+    1000  0:00 ./toto1
 3477  3478  3476  1999 pts/2  3476 S+    1000  0:00 ./toto1
 2427  3479  3479  2427 pts/3  3479 R+    1000  0:00 ps axj
```

La commande `top`

- *Rappel* : permet d'afficher les processus en cours d'exécution de façon dynamique
- Différentes informations affichées :
  - Le PID des processus
  - Les utilisateurs des processus
  - La priorité du processus
  - Mémoire utilisée
  - L'utilisation CPU
  - Le temps d'exécution
  - La commande
- Quelques options :
  - `top -u toto` : affiche uniquement les processus de l'utilisateur `toto`
  - `top -d 1.0` : modifie la fréquence de rafraîchissement

Le répertoire `proc` (1/2)

- Pseudo système de fichiers contenant un ensemble d'informations :
  ↪ Sur le système, l'architecture, la mémoire, etc.
  ↪ Les processus en cours d'exécution
- Exemples de fichiers :
  - `proc/cpuinfo` : informations sur le(s) processeur(s)
  - `proc/filesystems` : liste des systèmes de fichiers supportés
  - `proc/meminfo` : informations sur la mémoire utilisée
- Les répertoires :
  ↪ Un par processus en cours d'exécution

Le contenu du répertoire `/proc` et des fichiers qu'il contient peut être différent suivant les systèmes.

Le répertoire `proc` (2/2)

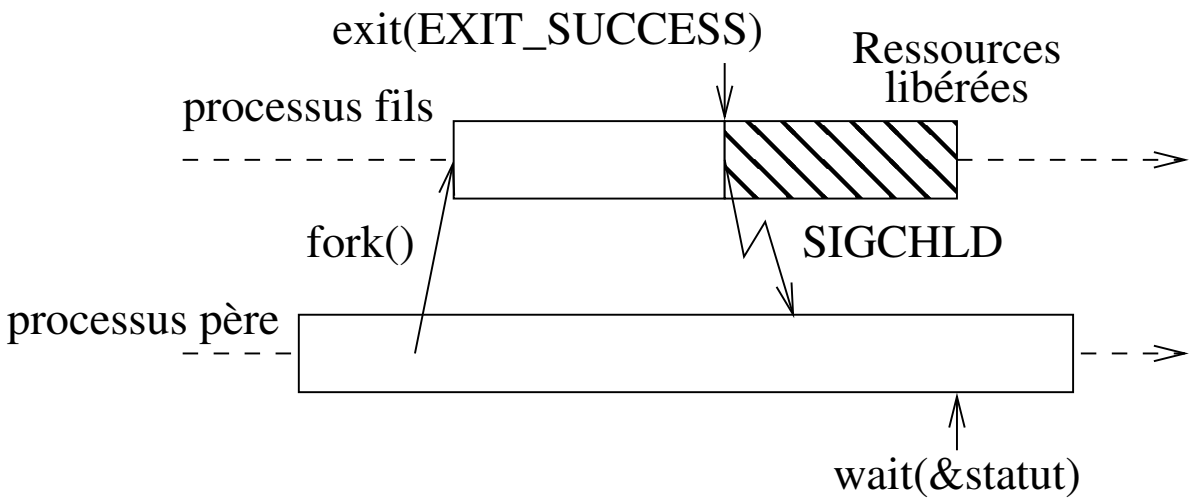
- Différents fichiers/répertoires dans les répertoires des processus :
  - `cmdline` : ligne de commande utilisée pour exécuter le programme
  - `cwd` : le répertoire courant du processus (sous forme de lien)
  - `environ` : les variables d'environnement
  - `exe` : lien vers le binaire
  - `fd` : répertoire contenant les descripteurs de fichiers ouverts

Exemple : `ls -al fd`

```
dr-x------ 2 rabat rabat  0 Jan 19 21:04 .
dr-xr-xr-x 9 rabat rabat  0 Jan 19 21:04 ..
lrwx------ 1 rabat rabat 64 Jan 19 21:04 0 -> /dev/pts/3
lrwx------ 1 rabat rabat 64 Jan 19 21:04 1 -> /dev/pts/3
lrwx------ 1 rabat rabat 64 Jan 19 21:04 2 -> /dev/pts/3
l-wx------ 1 rabat rabat 64 Jan 19 21:04 3 -> /home/rabat/Info0601/CM_04/toto.bin
```

Description générale

- 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



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

```c
/* 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 */
```

- 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()`

```c
/* 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

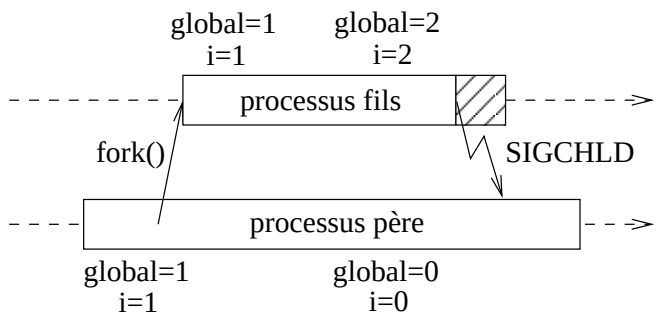
- Qu'est-ce que ça affichera ? (à part les PID...)

Exemple d'utilisation de `fork()`

```c
/* 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

- 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 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

```c
/* Includes */
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

```
...
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`

- 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)

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)

```
/* Includes */
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

- 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

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

```
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 (S3) (1/2)

- `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 !

Manipuler les variables d'environnements (S3) (2/2)

Inclusions

- *Inclusion* : `stdlib.h`

Exigence de macro pour `setenv` et `unsetenv`

```
_BSD_SOURCE || _POSIX_C_SOURCE >= 200112L
            || _XOPEN_SOURCE >= 600
```

Exigence de macro pour `putenv`

```
_SVID_SOURCE || _XOPEN_SOURCE
```

Chargement de processus : `execve`

- 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

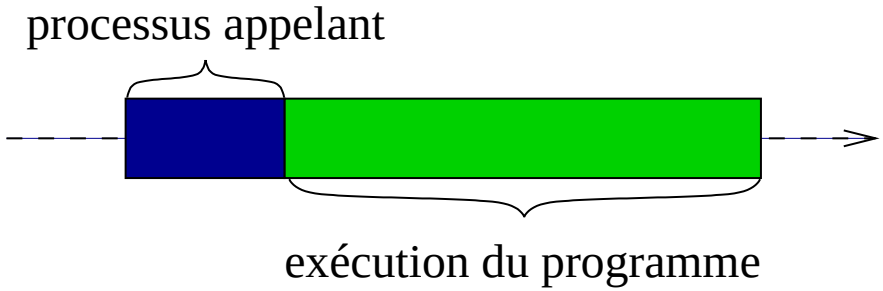
Frontaux de `execve` (S3)

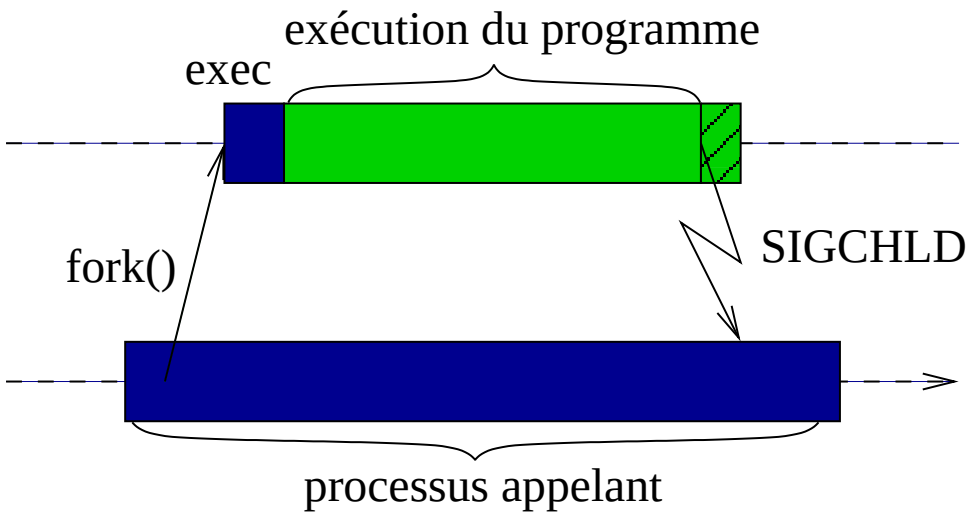
- `int execl(const char *path, const char *arg0, ..., NULL)`
- `int execlp(const char *file, const char *arg0, ..., NULL)`
- `int execle(const char *path, const char *arg0, ..., char *const envp[]) :`
- `int execv(const char *path, char *const argv[])`
- `int execvp(const char *file, char *const argv[])`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

Utilisation de `execve` conseillée !

Principes de fonctionnement

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

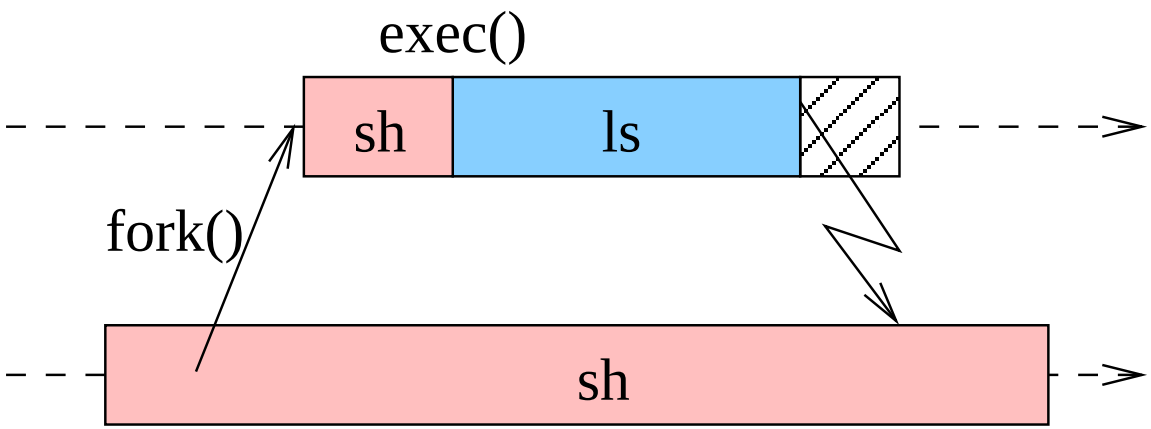




Pour garder la main : utiliser `fork`

Exemple : appel d'une commande par le shell

- 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



Et au niveau du processus ?

- 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)

```c
/* Code du programme 'helloworld' (sans inclusions) */
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 du programme 'execution' (sans inclusions) */
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)

```
/* 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);
}
```