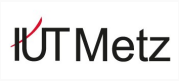
R3.05 - Programmation Système

Processus

Cyril Grelier

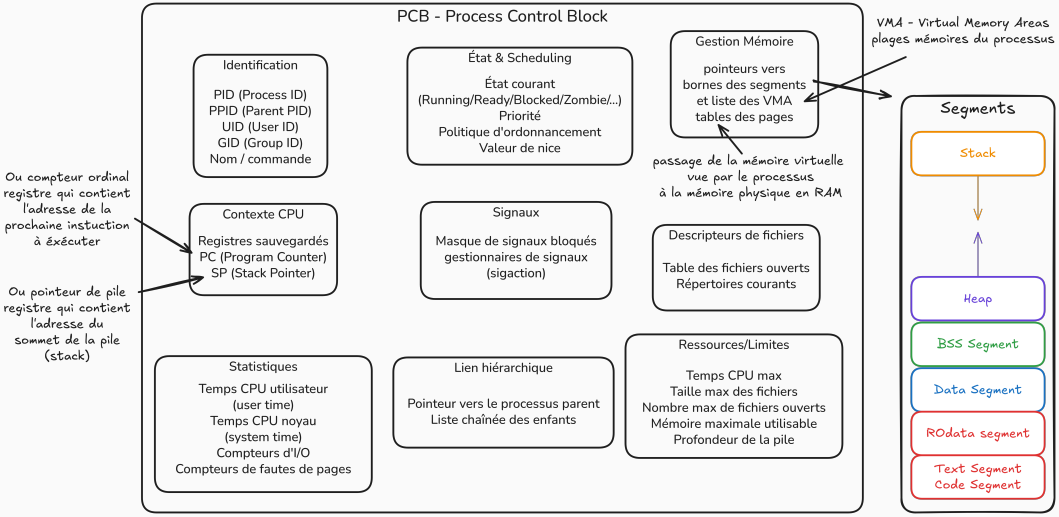
Université de Lorraine - IUT de Metz



Qu'est-ce qu'un processus ?

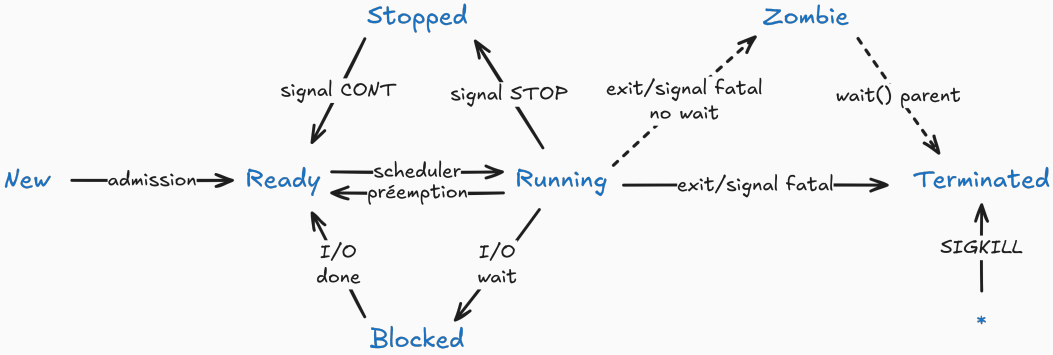
- **Processus** = instance d'un programme en exécution (plusieurs instances possibles)
- Analogie avec la POO : programme = classe ; processus = instance de la classe
- Unité gérée par l'OS, avec ressources dédiées :
  - **Mémoire privée** (VMA - Virtual Memory Areas)
  - IDs : **PID** (Process), **PPID** (Parent Process), **UID** (User)
  - **Descripteurs** de fichiers (0,1,2...)
  - **Contexte CPU** (PC - *Program Counter*, registres, état)
  - voir le `/proc/PID/...` du processus
  - ...
- Le noyau regroupe toutes les informations dans le **PCB** (*Process Control Block*)
- Le noyau garde une table des processus, pour chaque processus, un pointeur vers son PCB.

PCB - Process Control Block



Chaque processus possède son PCB

Cycle de vie d'un processus



- Arrêts par le noyau : **OOM killer**, limites ressources, signaux (Segfault, /0, ...).

Arbre des processus & outils

- Le PID 1 (`init`/`systemd`) adopte les orphelins.
- Visualiser :

```
$ pstree -p              # hiérarchie parent/enfant
$ ps                     # processus attachés au TTY courant
$ ps aux                 # tous les processus (+ infos)
```

`&` lance en arrière-plan, `fg` ramène au premier plan.

```
$ sleep 10 &
[1] 3816906
$ ps
  PID TTY   TIME CMD
3816906 pts/2 0:00 sleep
```

Créer/dupliquer un processus : fork() (processus lourd)

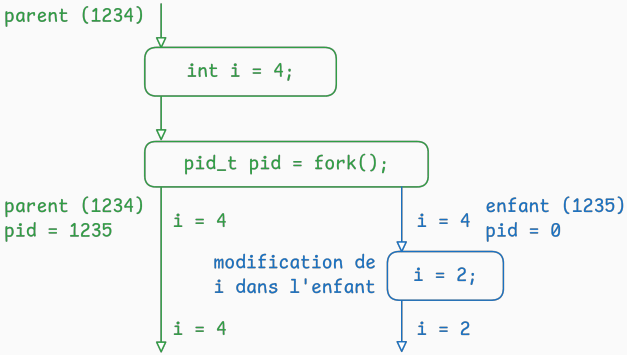
```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    printf("Avant le fork : PID = %d\n", getpid());
    pid_t pid = fork(); // Création d'un nouveau processus
    if (pid < 0) { // Si fork échoue
        perror("Erreur lors du fork"); exit(EXIT_FAILURE);
    } else if (pid == 0) { // Code exécuté par le processus enfant
        printf("Enfant : mon PID = %d, parent PID = %d\n", getpid(), getppid());
    } else { // Code exécuté par le processus parent
        printf("Parent : mon PID = %d, PID enfant = %d, parent PID = %d\n",
               getpid(), pid, getppid());
    }
    printf("Processus %d termine son exécution.\n", getpid());
}
```

```
Avant le fork : PID = 10
Parent : mon PID = 10, PID enfant = 11, PPID = 9
Processus 10 termine son exécution.
Enfant : mon PID = 11, PPID = 10
Processus 11 termine son exécution.
```

Mémoire copiée à l'écriture (COW - *Copy-On-Write*)

- Après fork, l'espace mémoire est **virtuellement** copié : pages partagées **en lecture seule**.
- Le PCB est copié et mis à jour pour le nouveau processus
- Lors d'une **écriture** dans une page, le noyau **duplique** la page.
- Garantit l'isolement parent/enfant à coût réduit.




```
La mémoire est dupliquée entre le parent est l'enfant
Si l'un modifie une variable déclarée avant le fork
elle ne sera pas modifiée dans les autres processus
Pour le parent, la variable pid = 1235
Pour l'enfant, la variable pid = 0
```

Remplacer un processus : exec*

- exec* remplace l'image mémoire ; le **PID ne change pas**.
- Variantes : l (liste), v (vecteur), p (PATH), e (environnement), fexecve (par FD) : execl, execv, execle, execve, execlp, execvp, execvpe, fexecve

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

int main(void){
  execl("/bin/ls","ls","-l",NULL);
  perror("execl"); // si on arrive ici, exec a échoué
  return 1;
}
```

Schémas classique : parent fait un `fork()` → l'enfant fait un `exec()` → le parent attend la fin de l'enfant avec `wait()`. Comme dans le terminal !

Synchronisation : wait/waitpid

```c
#include <sys/wait.h>

int status;
if (wait(&status) == -1) { /* perror */ }
if (WIFEXITED(status)) // W = Wait, if exited (s'il a terminé avec un return ou exit)
    printf("code=%d\n", WEXITSTATUS(status)); // code de retour
else if (WIFSIGNALED(status)) // if signaled (s'il a terminé à cause d'un signal)
    printf("signal=%d\n", WTERMSIG(status)); // numéro du signal ayant causé l'arrêt
```

- `waitpid(pid,&status,0)` pour attendre un enfant précis
- Dans l'enfant, utiliser `_exit()` (pas `exit()`, évite de vider 2 fois les buffeurs)
- **Zombie** : enfant fini mais pas de `wait` du parent ⇒ **Terminated** dès que le parent fait le `wait`
- **Orphelin** : parent terminé ⇒ adopté par PID 1 (`init`/`systemd`) qui fera le `wait`

Ordonnanceur, priorité, limites

Politiques : SCHED_OTHER (CFS - *Completely Fair Scheduler*), SCHED_FIFO, SCHED_RR.

```
$ chrt -p <pid> # politique/priorité d'un PID avec SCHED_FIFO et SCHED_RR
$ sudo chrt -r 20 ./prog # lancer en temps réel RR priorité 20
```

Nice : de -20 (très prioritaire) à $+19$ (très gentil) (avec SCHED_OTHER).

```
$ nice -n 10 ./long_calcul
$ renice -5 -p <pid>
$ ps -o pid,comm,pri,ni -p <pid>
```

Limites (bash) :

```
$ ulimit -a
$ ulimit -t 5   # temps CPU
$ ulimit -n 64  # nb fichiers ouverts
```

Explorer /proc

```
$ echo $$              # PID du shell
12345
$ ls /proc/$$
attr cmdline cwd environ fd/ maps status task ...
$ head -n 5 /proc/$$/status
Name: bash
State: S (sleeping)
Pid:   12345
PPid:  678
```

/proc/<pid>/fd : descripteurs ouverts ; maps : mappage mémoire.

/proc/self : alias vers le processus courant

Résumé pratique : fork → exec → wait

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

int main(void){
  pid_t pid = fork();
  if (pid < 0) { perror("fork"); exit(1); }
  if (pid == 0) { // enfant
    execl("/bin/ls","ls","-l",NULL);
    perror("execl");
    _exit(1);
  } else { // parent
    int st=0;
    waitpid(pid,&st,0);
    if (WIFEXITED(st))
        printf("retour=%d\n", WEXITSTATUS(st));
  }
}
```

Toujours vérifier les retours d'appels système et `perror()` en cas d'échec.

IPC - *Inter Process Communication*

- Chaque **processus** a sa mémoire virtuelle isolée.
- Pour coopérer $\Rightarrow$ besoin de **mécanismes IPC**.
- Principaux mécanismes sous Linux/POSIX :
  - **Signaux** interruption logicielle
  - **Pipes** (tubes) entre processus apparentés
  - **FIFOs** (tubes nommés) pour processus non apparentés
  - **Files de messages**, **mémoire partagée**, **sémaphores** (hors programme)
  - **Sockets** (voir chapitre réseaux)

Signaux : notions & exemples

- Mécanisme asynchrone de communication inter-processus.
- Envoie une interruption logicielle à un processus

```
$ kill -L
1) SIGHUP    2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
6) SIGABRT   7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV 12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT       17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
... jusqu'à 64
```

- Exemple, lancer gedit (ou autre programme) en arrière plan puis le tuer :

```
$ gedit fichier.txt & # 104827 est le pid du processus
[1] 104827
$ kill -SIGINT 104827   # interrompt gedit
```

Installer un handler avec sigaction

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

static volatile sig_atomic_t stop = 0;
static void on_sigint(int sig){ (void)sig; stop = 1; }

int main(void){
  struct sigaction sa = {0};
  sa.sa_handler = on_sigint;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = SA_RESTART;
  sigaction(SIGINT, &sa, NULL);

  printf("PID=%d, Ctrl+C pour quitter\n", getpid());
  while(!stop) pause();
}
```

Pipes anonymes

- Canal de communication **unidirectionnel**.
- Entre deux processus apparentés (*ex. parent/enfant*).
- Création avec :

```c
#include <unistd.h>
int pipe(int fd[2]);
/* fd[0] = lecture, fd[1] = écriture */
```

Remarques

- Si besoin de communication bidirectionnelle → deux pipes.
- Fermer les extrémités inutiles (sinon EOF jamais envoyé).

Exemple : pipe parent → enfant

```c
int fd[2];
pipe(fd);
pid_t pid = fork();

if (pid == 0) {
  // Enfant
  close(fd[1]);              // ferme écriture
  char buf[64];
  ssize_t n = read(fd[0], buf, sizeof(buf)-1);
  buf[n] = '\0';
  printf("Enfant a reçu : %s\n", buf);
  close(fd[0]);
} else {
  // Parent
  close(fd[0]);              // ferme lecture
  const char *msg = "Bonjour !";
  write(fd[1], msg, strlen(msg));
  close(fd[1]);
}
```

Pipes nommés (FIFO)

- Tube stocké dans le **système de fichiers**.
- Permet communication entre processus **non apparentés**.
- Création avec :

```c
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

Propriétés

- open(O_RDONLY) bloque jusqu'à un écrivain.
- open(O_WRONLY) bloque jusqu'à un lecteur.
- En non-bloquant (O_NONBLOCK) :
  - Lecture : retourne immédiatement.
  - Écriture : échoue si pas de lecteur.
- EOF : quand tous les écrivains ferment.
- Suppression avec unlink("canal").

Exemple FIFO

En bash :

```
mkfifo canal
cat < canal # Terminal 1
echo "coucou" > canal # Terminal 2
# Terminal 1 affiche "coucou"
```

En C :

```c
const char *path = "canal";
if (mkfifo(path, 0666) == -1)
    perror("mkfifo");

// bloque si pas de lecteur
int w = open(path, O_WRONLY);

const char *msg = "Hello FIFO\n";
write(w, msg, strlen(msg));
close(w);

// unlink(path); // pour supprimer la FIFO
```

```c
const char *path = "canal";
if (mkfifo(path, 0666) == -1 && errno != EEXIST)
    perror("mkfifo");

// bloque si pas d'écrivain
int r = open(path, O_RDONLY);

char buf[4096];
ssize_t n;
while ((n = read(r, buf, sizeof buf)) > 0) {
  write(STDOUT_FILENO, buf, n);
}
close(r);
```