Gestion des Processus

Aloÿs Dufour

ATER, LIPN équipe LoCal
Université Paris-Nord XIII

30 janvier 2026

Gestion des processus

Definition

Un processus est un programme en cours d'exécution. On en distingue deux types :

les processus système (démons) : assurent des services généraux accessibles à tous les utilisateurs du système. Le propriétaire est `root` et il n'est rattaché à aucun terminal,

les processus utilisateur : dédiés à l'exécution d'une tâche particulière. Le propriétaire est l'utilisateur qui l'exécute et il est sous le contrôle du terminal à partir duquel il a été lancé.

Gestion des processus

Création

- Toute exécution d'un programme déclenche la création d'un processus dont la durée de vie est la durée d'exécution du programme.
- Le système alloue à chaque processus un numéro d'identification unique : PID (Process IDentifier).
- Tout processus est créé par un autre processus : son père.

Gestion des processus

Exécution d'une commande : 5 modes d'exécution (sous Unix) :

- mode interactif : commande lancée à partir d'un terminal. Le contrôle du terminal n'est rendu à l'utilisateur qu'à la fin de l'exécution de la commande. `<ctrl-c>` : interrompre la commande `<ctrl-z>` : suspendre la commande
- mode en arrière plan : permet de rendre immédiatement le contrôle à l'utilisateur (`command &`). Si le terminal est fermé, la commande en arrière plan est interrompue automatiquement (pour éviter ce problème `nohup command &`).

Gestion des processus

Exécution d'une commande :

- mode différé : `at` permet de déclencher l'exécution d'une commande à une date fixée.
  ex : `at 16:50 10/06/08 < commande`.
  (`at -l` pour lister et `at -r` pour supprimer)
- mode périodique : tâche exécutée de manière périodique, grâce à la `crontab` par ex.
- mode batch : permet de placer une commande dans une file d'attente.

Gestion des processus — La commande ps

- Visualiser les processus avec la commande : `ps (options)`, les options les plus intéressantes
  - `-e` : affichage de tous les processus
  - `-f` : affichage détaillé
- exemple : `ps -ef`

```
UID      PID PPID C STIME    TTY   TIME COMMAND
root     1   0    0 Dec 6    ?     1:02 init
...
jean     319 300  0 10:30:30 ?     0:02 /usr/dt/bin/dtsession
olivier 321 319  0 10:30:34 ttyp1 0:02 csh
olivier 324 321  0 10:32:12 ttyp1 0:00 ps -ef
```

Gestion des processus — La commande ps

Signification des différentes colonnes

`UID`	Utilisateur qui a lancé le processus
`PID`	Identifiant du procesuss
`PPID`	Identifiant du processus parent
`C`	Utilisation du processeur (`%cpu = cputime/realtime`)
`STIME`	Date de lancement
`TTY`	Terminal contrôlant le processus
`TIME`	Durée cumulative de traitement
`COMMAND`	Nom de la commande avec ses arguments

Pour voir les process d'un seul utilisateur : `ps -u olivier`

Gestion des processus — La commande ps

Signification des différentes colonnes

`UID`	Utilisateur qui a lancé le processus
`PID`	Identifiant du procesuss
`PPID`	Identifiant du processus parent
`C`	Utilisation du processeur (`%cpu = cputime/realtime`)
`STIME`	Date de lancement
`TTY`	Terminal contrôlant le processus
`TIME`	Durée cumulative de traitement
`COMMAND`	Nom de la commande avec ses arguments

Pour voir les process d'un seul utilisateur : `ps -u olivier`
Naviguer les processus de manière interactive (TUI) et voir leur évolution : `htop`

Gestion des processus — Les signaux

Il est possible d'agir sur le déroulement d'un *p* en lui envoyant un signal. POSIX standardise un certain nombre de signaux dont :

- `SIGINT` (signal 2) : interrompre l'exécution de *p*, (même signal que `<ctrl+c>`),
- `SIGKILL` (signal 9) : arrêt violent de l'exécution de *p* (ne peut pas être ignoré),
- `SIGTERM` (signal 15) : arrêt de l'exécution de *p*,
- `SIGTSTP` (signal 19) : suspendre temporairement l'exécution de *p*,
- `SIGCONT` (signal 18) : reprendre l'exécution de *p* précédemment suspendu par l'envoi d'un signal `SIGTSTP`.

```
man 7 signal
```

Gestion des processus

Un signal peut être envoyé par :

- le système (ex : signaux d'erreur),
- un autre processus,
- l'utiliateur :
  - soit l'utilisateur tape des caractères provoquant l'envoi d'un signal au processus en cours d'exécution sur le terminal (ex : `<ctrl-z>` pour `SIGTSTP`, `<ctrl-c>` pour `SIGINT`),
  - soit l'utilisateur utilise la commande `kill` pour envoyer un signal à un ou plusieurs processus lorsqu'il n'a pas accès au terminal de rattachement des processus ou lorsque ces derniers sont exécutés en arrière plan :
    `$ kill -signal PID`
    `signal` : nom symbolique ou num.

Gestion des processus — États d'un processus

1. s'exécute en mode utilisateur

Gestion des processus — États d'un processus

1. s'exécute en mode utilisateur
2. s'exécute en mode noyau

Gestion des processus — États d'un processus

1. s'exécute en mode utilisateur
2. s'exécute en mode noyau
3. ne s'exécute pas mais est éligible (prêt à s'exécuter)

Gestion des processus — États d'un processus

1. s'exécute en mode utilisateur
2. s'exécute en mode noyau
3. ne s'exécute pas mais est éligible (prêt à s'exécuter)
4. est endormi en mémoire centrale

Gestion des processus — États d'un processus

1. s'exécute en mode utilisateur
2. s'exécute en mode noyau
3. ne s'exécute pas mais est éligible (prêt à s'exécuter)
4. est endormi en mémoire centrale
5. est prêt mais le swappeur doit le transférer en mémoire centrale pour le rendre éligible.

Gestion des processus — États d'un processus

1. s'exécute en mode utilisateur
2. s'exécute en mode noyau
3. ne s'exécute pas mais est éligible (prêt à s'exécuter)
4. est endormi en mémoire centrale
5. est prêt mais le swappeur doit le transférer en mémoire centrale pour le rendre éligible.
6. est endormi en zone de swap.

Gestion des processus — États d'un processus

1. s'exécute en mode utilisateur
2. s'exécute en mode noyau
3. ne s'exécute pas mais est éligible (prêt à s'exécuter)
4. est endormi en mémoire centrale
5. est prêt mais le swappeur doit le transférer en mémoire centrale pour le rendre éligible.
6. est endormi en zone de swap.
7. passe du mode noyau au mode utilisateur mais est préempté et a effectué un changement de contexte.

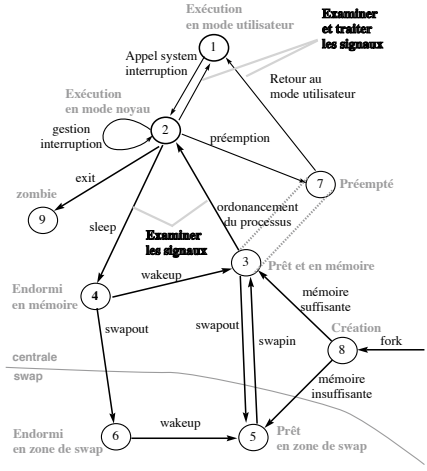
Gestion des processus — États d'un processus

1. s'exécute en mode utilisateur
2. s'exécute en mode noyau
3. ne s'exécute pas mais est éligible (prêt à s'exécuter)
4. est endormi en mémoire centrale
5. est prêt mais le swappeur doit le transférer en mémoire centrale pour le rendre éligible.
6. est endormi en zone de swap.
7. passe du mode noyau au mode utilisateur mais est préempté et a effectué un changement de contexte.
8. naissance d'un processus, ce processus n'est pas encore prêt et n'est pas endormi, c'est l'état initial de tout processus.

Gestion des processus — États d'un processus

1. s'exécute en mode utilisateur
2. s'exécute en mode noyau
3. ne s'exécute pas mais est éligible (prêt à s'exécuter)
4. est endormi en mémoire centrale
5. est prêt mais le swappeur doit le transférer en mémoire centrale pour le rendre éligible.
6. est endormi en zone de swap.
7. passe du mode noyau au mode utilisateur mais est préempté et a effectué un changement de contexte.
8. naissance d'un processus, ce processus n'est pas encore prêt et n'est pas endormi, c'est l'état initial de tout processus.
9. *zombie* : le processus vient de réaliser un `exit`, il apparaît uniquement dans la table des processus où il est conservé le temps pour son processus père de récupèrer le code de retour...

Gestion des processus — États d'un processus



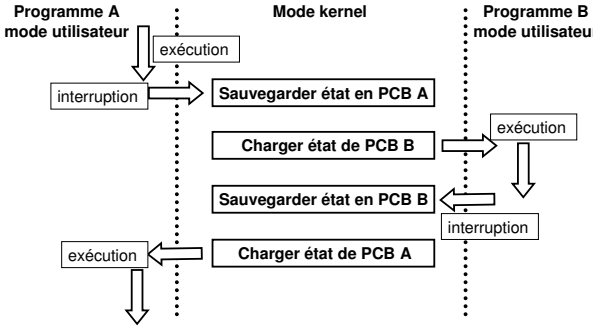
Gestion des processus — Implémentation des processus

Descripteur de Processus (PCB)

- Le PID du processus
- l'état du processus
- son compteur ordinal
- son allocation mémoire
- les fichiers ouverts
- les valeurs contenues dans les registres du processeur

tout ce qui doit être sauvegardé lorsque l'exécution d'un processus est suspendue

Gestion des processus — Changement de contexte



Fonctions Posix pour la gestion de processus

POSIX définit un nombre relativement petit d'appels système pour la gestion de processus :

- Création de processus (fils)

  ```
  pid_t fork()
  ```

- Fonction permettant à un processus d'exécuter un autre programme

  ```
  int execl(), execlp(), execvp(), execle(), execv()
  ```

- Attendre la terminaison d'un processus

  ```
  pid_t wait()
  ```

- Finir l'exécution d'un processus

  ```
  void exit()
  ```

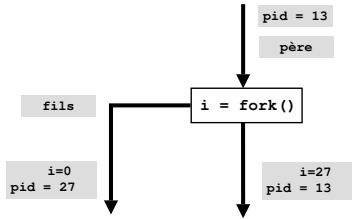
- Récupérer le PID ou le PPID

  ```
  pid_t getpid()
  pid_t getppid()
  ```

Gestion des processus — Création de processus

```c
#include<unistd.h>
#include<sys/types.h>
pid_t fork(void);
```

- L'image mémoire est la même pour le père et le fils
- Valeur de retour du `fork()`
  - 0 pour le processus fils
  - Strictement positive pour le processus père : PID du fils
  - Négative si la création de processus a échoué



Création de processus — Exemple 1

```c
#include <unistd.h>
#include <sys/types.h>
int main()
{
    pid_t pidfils;
    pidfils = fork();
    if (pidfils == 0)
        printf("Je suis le fils avec pid%d\n",getpid());
    else
        if (pidfils > 0)
            printf("Je suis le père avec pid%d\n", getpid());
        else
            printf("Erreur dans la création du fils\n");
}
```

Création de processus — Exemple 2

Combien de processus sont créés par le programme suivant ?

```
int main() {
    fork() || (fork() && fork());
    exit(EXIT_SUCCESS);
}
```

De même avec la ligne : `fork() && (fork() || fork());`.

Création de processus — Exemple 3

Combien de processus sont créés par le programme suivant ?

```
int main()
{
    int i, n = 5;
    int childpid;
    for(i=1; i<n; i++)
    {
        if ((childpid=fork()) <= 0)
            break;
        printf ("Processus %d avec pere %d , i=%d\n", getpid(), getppid(), i);
    }
    return 0 ;
}
```

Et si on remplace `if(... <= 0)` par `if(... < 0)` ?

Création de processus — Exemple 4 (modification de variable)

```c
int main(void)
{
    pid_t p ;
    int a = 20;
    switch (p = fork()) {
        case -1:
            perror("Le fork a échoué !");
            break;
        case 0:
            printf("Ici processus fils, de PID%d.\n", getpid());
            a += 10;
            break;
        default:
            printf("Ici processus père, de PID%d.\n", getpid());
            a += 100;
    }
    printf("Fin du processus%d avec a = %d.\n", getpid(), a);
    return 0;
}
```

Création de processus — Exemple 5 (processus orphelin)

```c
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    int i, j, k, n = 5;
    pid_t pidfils;
    for (i=1; i<n; i++) {
        pidfils = fork();
        if (pidfils > 0) /* c'est le père */
            break;
        printf ("Processus %d avec pere %d\n", getpid(), getppid());
    }
    /* Pour faire perdre du temps au processus */
    for (j=1; j<100000; j++)
        for(k=1; k<1000; k++);
```

Création de processus — Exemple 6 (processus zombie)

```c
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    pid_t pid;
    pid = fork();
    if (pid > 0) {
        /* Père : dormir 30 secondes */
        sleep(30);
    } else {
        /* Fils : quitter immediatement */
        exit(0);
    }
    return 0
}
```

WAIT(), WAITPID() ET EXIT()

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

Ces appels système permettent au processus père d'attendre la fin d'un de ses processus fils et de récupérer son status de fin (`WEXITSTATUS(status)`). Ainsi, un processus peut synchroniser son exécution avec la fin de son processus fils en exécutant l'appel système `wait()`. La syntaxe de l'appel système est :

```
pid = wait(status);
```

où `pid` est l'identifiant du processus fils et `status` est l'adresse dans l'espace utilisateur d'un entier qui contiendra le status de `exit()` du processus fils.

Famille des appels exec()

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

Un processus fils créé peut remplacer son code de programme par un autre programme. Tous les appels système exec remplacent le processus courant par un nouveau processus construit à partir d'un fichier ordinaire exécutable.

Famille des appels exec()

- `execl()` permet de passer un nombre fixé de paramètres au nouveau programme.
- `execv()` permet de passer un nombre libre de paramètres au nouveau programme via `char *argv[]`.
- `execle()`, même fonctionnement qu'`execl()` avec en plus, un argument `char *envp[]` qui représente l'environnement.
- `exelp()`, arguments et action identiques à celles d'`execl()`, mais la différence vient du fait que si le nom du fichier n'est pas un chemin absolu, le système utilisera la variable `PATH` listant les répertoires dans lesquels se trouvent les exécutables accessibles depuis n'importe quel répertoire.
- `exevp()`, arguments et action identiques à celles d'`execv()`, mais la différence vient du fait que si le nom de fichier n'est pas un chemin absolu, la commande utilise les répertoires spécifiés dans le `PATH`.

La convention Unix veut que chaque chaîne ait la forme `nom=valeur`. Ainsi, les arguments pour `execv()` peuvent être passés, par exemple :

```
char *arguments[4] ;
arguments[0] = "/bin/ls";
arguments[1] = "-l";
arguments[2] = "/etc";
arguments[3] = "NULL";
execv("/bin/ls", arguments);
```

Création de processus — Exemple (exec + wait)

```c
int main(int argc, char *argv[])
{
    pid_t pidfils ;
    /* Liste d'arguments pour la comande "ls" */
    char args[] = {"ls", "ls", argv[1], NULL};
    /* Dupliquer le processus */
    pidfils = fork();
    if (pidfils != 0) {
        /* Il s'agit du père */
        waitpid(pidfils, NULL, NULL);
        printf("Programme principal termine\n");
        return 0;
    } else {
        /* Exécuter programme avec les arguments a partir du PATH */
        execvp("ls", args);
        /* Retourner en cas d'erreur */
        perror("Erreur dans execvp");
        exit(1);
    }
    return 0;
}
```

Création de processus

```c
#define N_DFT 2
#define P_DFT 3
#define FMT_PROC "pid : %d; ppid : %d; n : %d; p : %d\n"

int main(int argc, char *argv[])
{
    int n = argc < 2 ? N_DFT : atoi(argv[1]);
    int p = argc < 3 ? P_DFT : atoi(argv[2]);
    int i;

    if (p == 0) {
        printf("Sortie 1\n");
        return 0;
    }

    for (i = 0; i < n; ++i)
        switch (fork()) {
            case -1:
                perror("fork");
                return 1;
            case 0:
                printf(FMT_PROC, getpid(), getppid(), n, p);
                char str_n[11], str_p[11];
                sprintf(str_n, "%d", n);
                sprintf(str_p, "%d", p - 1);
                execl("encore", "encore", str_n, str_p, NULL
                printf("Sortie 2\n");
                return 0;
        }

    while (wait(NULL) != -1)
        ;
    printf("Sortie 3\n");
    return 0;
}
```

1. Dans le cas où le programme est lancé de la façon suivante : `$ ./encore` dessiner l'arborescence des processus et donner le nombre de `Sortie 1`, `Sortie 2` et `Sortie 3` qui seront affichés.
2. Soit $u_{n,p}$ le nombre de processus créés par ce programme lorsque les arguments entiers naturels n et p lui sont transmis. Donner la relation entre $u_{n,p}$ et $u_{n,p-1}$ lorsque $p > 0$. Puis calculer $u_{n,p}$. En déduire le nombre d'affichages de `Sortie 1`, `Sortie 2` et `Sortie 3` en fonction de n et p.