Rapport SYSG5 : Controle de jobs

Filipe PEREIRA MARTINS - 58093

Novembre 2023

Table des matières

1 Introduction

Le contrôle des jobs dans un système d'exploitation Linux permet l'exécution simultanée de plusieurs commandes avec un contrôle précis. Cette fonctionnalité permet d'assigner un job en "foreground" pendant que d'autres s'exécutent en "background", simplifiant ainsi la gestion des tâches. On peut arrêter, déplacer et surveiller les jobs en cours, offrant une solution pratique pour effectuer diverses opérations sans ouvrir de nouveaux terminaux.
Cette recherche a été initiée pour approfondir la compréhension des mécanismes régissant le contrôle des jobs dans l'environnement Linux. Le rapport est structuré en plusieurs sections, chacune se concentrant sur un aspect particulier du contrôle des jobs. L'objectif principal est de présenter une vue d'ensemble exhaustive de cette fonctionnalité, en débutant par ses fondements pour ensuite explorer des concepts plus avancés.

2 Aperçu

Le contrôle de jobs sous Linux repose sur l'organisation des processus en groupes et sessions. Les groupes de processus permettent de gérer efficacement les tâches, regroupant des processus ayant des identifiants de groupe de processus (PGID) communs. Chaque groupe a un leader, créant le groupe et définissant son PGID. La durée de vie du groupe commence avec sa création et se termine lorsque le dernier processus le quitte.
Les groupes de processus sont ensuite regroupés au sein de sessions, définies par des identifiants de session (SID). Une session est initiée par un leader de session, devenant le contrôleur du terminal et attribuant un terminal de contrôle à tous les processus de la session.
Dans le contexte du contrôle de jobs, le terminal de contrôle est crucial. Lorsqu'un utilisateur se connecte, le shell de connexion devient le leader de session et de groupe, contrôlant le terminal. Chaque commande/job lance un nouveau groupe de processus, avec la possibilité d'être en foreground ou background. Le leader de session reçoit des signaux du terminal, comme SIGHUP en cas de déconnexion.
Ce système offre une gestion flexible des jobs, avec la création de groupes de processus et de sessions pour optimiser le contrôle des interactions avec le terminal.

3 Groupes de Processus

Un groupe de processus est une collection d'un ou plusieurs processus qui partagent un identifiant de groupe de processus (PGID) commun. L'identifiant de groupe de processus (PGID) est un numéro de type *pid_t*, similaire à l'identifiant de processus (PID).
Chaque groupe de processus est dirigé par un chef de groupe de processus, qui est le processus responsable de la création du groupe et dont l'identifiant de processus (PID) devient l'identifiant de groupe de processus (PGID) du groupe. Lorsqu'un nouveau processus est créé, il hérite automatiquement de l'identifiant de groupe de processus (PGID) de son processus parent.

3.1 L'appel système *setpgid()*

La création d'un nouveau groupe de processus, essentielle dans le contrôle des jobs, est réalisée via l'appel système *setpgid()* . Cette fonction permet de changer le groupe de processus d'un processus existant ou de créer un groupe totalement nouveau.

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

L'appel système *setpgid()* modifie le groupe de processus d'un processus existant ou crée un nouveau groupe. Les paramètres *pid* et *pgid* déterminent le processus et le nouveau PGID respectivement. Si ces paramètres spécifient le même processus, un nouveau groupe est créé, avec le processus spécifié en tant que leader.
Une restriction importante est que le PGID d'un enfant ne peut pas être modifié après un *exec()*, déclenchant une erreur EACCES si tenté.

3.1.1 Création d'un nouveau groupe

Voici un petit code qui démontre la création d'un nouveau groupe à l'aide de *setpgid()* :

Listing 1 – creationGroupe.c

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

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Enfant avant creation du groupe - PID: %d, PGID: %d\n",
            getpid(), getpgrp());
```

```
        setpgid(0, 0); // Creer un nouveau groupe de processus avec le
            processus enfant comme leader
        printf("Enfant apres creation du groupe - PID: %d, PGID: %d\n",
            getpid(), getpgrp());
    } else if (pid > 0) {
        // Code du processus parent
        sleep(1); // Donner une chance au processus enfant de s'executer
            et de creer le nouveau groupe
        printf("Parent - PID: %d, PGID: %d\n", getpid(), getpgrp());
    } else {
        perror("Erreur lors de la creation du processus");
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

3.1.2 Contraintes dans le contrôle de jobs

La restriction interdisant à un processus de modifier l'identifiant du groupe de processus (PGID) de l'un de ses enfants après qu'il ait effectué un exec() influence la conception des shells de contrôle de jobs.
Il est essentiel de souligner que tous les processus d'un job (une commande ou un pipeline) doivent être regroupés dans un unique groupe de processus. Cela permet au shell d'envoyer simultanément des signaux de contrôle de job à tous les membres du groupe de processus. Naturellement, cette étape doit être réalisée avant l'émission de tout signal de contrôle de tâche !
Ainsi, chaque job (processus enfant) doit être assigné à un groupe de processus avant l'exécution d'un programme, car le programme lui-même n'a pas connaissance des modifications de l'identifiant du groupe de processus.
Pour chaque processus dans la tâche, soit le shell (parent), soit le job (enfant) pourrait utiliser setpgid() pour modifier l'identifiant du groupe de processus du job (enfant). Toutefois, en raison de l'imprévisibilité de l'ordonnanceur après un fork(), il n'est pas garanti que le shell changera l'identifiant du groupe de processus du nouveau job avant que ce dernier n'effectue un exec(). De même, on ne peut pas être assuré que le job (enfant) modifiera son identifiant de groupe de processus avant que le shell (parent) ne tente d'envoyer des signaux de contrôle de job. Par conséquent, les shells de contrôle de job sont conçus pour que le shell lui-même et le job (processus enfant) appellent tous deux setpgid() afin de modifier l'identifiant du groupe de processus du job à la même valeur immédiatement après un fork(). Le parent ignore toute occurrence de l'erreur EACCES lors de l'appel à setpgid().

3.2 L'appel système *getpgrp()*

Chaque processus possède un identifiant de groupe de processus qui définit le groupe de processus auquel il appartient (PGID). Un nouveau processus hérite de l'identifiant de groupe de processus de son parent. Un processus peut obtenir son identifiant de groupe de processus en utilisant getpgrp(). Si la valeur renvoyée par getpgrp() correspond à l'ID du processus appelant, cela signifie que ce processus est le leader de son groupe de processus.

```
#include <unistd.h>

pid_t getpgrp(void);
```

3.2.1 Obtenir le groupe

Voici un petit code qui démontre comment obtenir le groupe d'un processus à l'aide de *getpgrp()*.

Listing 2 – obtenirGroupe.c

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

int main(int argc, char * argv[]){
  pid_t pid, pgid; // PID et PGID du process courrant
  pid_t ppid, ppgid; // PID et PGID du parent
  // Process courrant
  pid = getpid();
  pgid = getpgrp();
  // Process Parent
  ppid = getppid();
  ppgid = getpgid(ppid);
  // Print PGID et PID du process courrant + parent
  printf("(Process courrant) pid:%d pgid:%d\n", pid, pgid);
  printf("(Process parent) ppid:%d pgid:%d\n", ppid, ppgid);

  return 0;
}
```

4 Sessions

Une session est un regroupement de plusieurs groupes de processus. La participation d'un processus à une session est déterminée par son identifiant de session (SID). Le leader de session est le processus responsable de la création d'une nouvelle session et dont sont identifiant (PID) devient l'identifiant de session (SID) de cette session (généralement le shell). Lorsqu'un nouveau processus est créé, il hérite automatiquement de l'identifiant de session (SID) de son processus parent.

4.1 L'appel système *setsid()*

L'appel système *setsid()* permet de créer une nouvelle session et définir le processus appelant comme le leader de cette session. Le PGID et SID deviennent le même que le PID du processus appelant.

```
#include <unistd.h>

pid_t setsid(void);
```

Il est important de noter que, d'une part, un leader de groupe de processus existant ne peut pas créer une nouvelle session. D'autre part, lorsqu'un processus fait l'appel à *setsid()*, il perd toute connexion préexistante au terminal de contrôle.
Le code ci-dessous, illustre la création d'une nouvelle session, pour démontrer que cette nouvelle session n'a plus d'accés au terminal de contrôle, le programme essaye d'ouvrir le terminal de contrôle associé à la session. Une éxécution de ce programme entraine un résultat similaire à celui-ci :

```
$ ./creationSession
Avant nouvelle session : PID=1023, PGID=1022, SID=944
Apres nouvelle session : PID=1023, PGID=1023, SID=1023
open /dev/tty : No such device or address
```

Listing 3 – creationSession.c

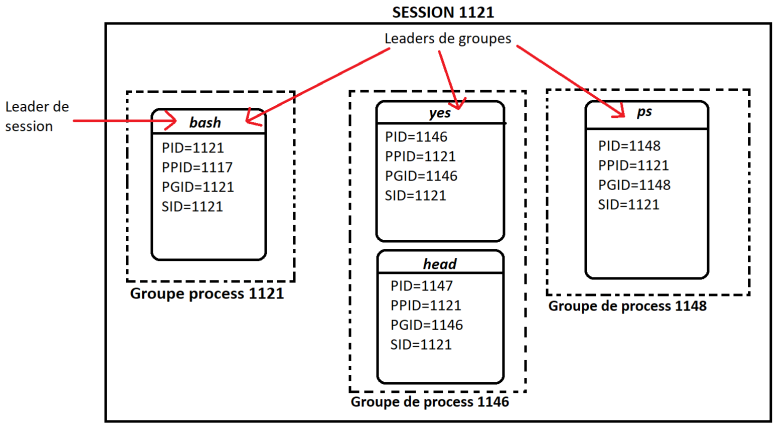
```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(void) {

    if (fork() != 0) // Exit du parent
        exit(0);
```

```
    printf("Avant nouvelle session : PID=%ld, PGID=%ld, SID=%ld\n",
        (long) getpid(),
        (long) getpgrp(), (long) getsid(0));

    if (setsid() == -1)
        perror("setsid");
        exit(-1);

    printf("Avant nouvelle session : PID=%ld, PGID=%ld, SID=%ld\n",
        (long) getpid(),
        (long) getpgrp(), (long) getsid(0));

    if (open("/dev/tty", O_RDWR) == -1) // Ouvrir terminal de controle
        associe
        perror("open /dev/tty");
        exit(-1);

    exit(0);
}
```

5 Relations Groupes/Sessions

Afin de clarifier les concepts de groupes de processus et de sessions, l'exemple de code shell ci-dessous offre une opportunité pratique pour illustrer leur interaction.

```
$ echo $$ # PID du shell
1121
$ yes "data" | head -n 1000000000 > /dev/null & # 2 process en background
[1] 1023
$ ps -o 'pid,ppid,pgid,sid,cmd' # 1 process en foreground
  PID    PPID    PGID     SID CMD
 1121    1117    1121    1121 bash
 1146    1121    1146    1121 yes data
 1147    1121    1146    1121 head -n 1000000000
 1148    1121    1148    1121 ps -o 'pid,ppid,pgid,sid,cmd'
```



6 Terminales de contrôle

Un terminal de contrôle est un terminal spécial associé à une session de processus. Il sert de point de communication principal entre une session et l'utilisateur, permettant aux processus de cette session d'interagir avec l'utilisateur via le terminal. Tous les processus d'une session peuvent avoir un seul et unique terminal de contrôle.

6.1 Association à un terminal de contrôle

Comme expliqué précedemment, lorsqu'une nouvelle session est créée, le processus leader de cette session n'a pas encore de connexion au terminal de contrôle. En d'autres termes, à sa création, une session ne dispose pas encore d'un terminal de contrôle associé.
Pour qu'une session puisse établir une connexion avec un terminal de contrôle, le processus leader de cette session doit ouvrir un nouveau terminal de contrôle. Il est essentiel de noter que ce terminal ne doit en aucun cas être déjà associé à une autre session.
Lorsque le leader de la session réussit à établir cette connexion avec le terminal, il devient ce que l'on appelle le "processus contrôleur". Les détails sur les implications de ce rôle seront expliqués dans la suite.
Tous les processus dans une session héritent de l'accès du terminal de contrôle. Habituellement, les nouvelles sessions sont créées par le programme de login du système et le leader de la session est le processus exécutant le shell de login de l'utilisateur.

7 Le signal SIGHUP

Lorsqu'une session (c'est-à-dire le processus contrôleur) perd son terminal de contrôle ou le processus contrôleur se termine (fermeture de la fenêtre du terminal par exemple), tous les processus de la session perdent également leur association à ce terminal de contrôle.
Pour les informer de cet évènement, le noyau envoie un signal (SIGHUP) au processus contrôleur, qui lui-même va créer une réaction en chaine en envoyant SIGHUP à tous les groupes de processus que lui-même a créé, à noter que ce signal ne sera pas envoyé à des groupes de processus (jobs) qui n'ont pas été créées par le processus contrôleur !
Le signal SIGHUP, par défaut termine un programme, mais celui-ci peut bien sur créer un handler, et continuer à s'exécuter (pour faire des opérations de nettoyage par exemple). Le signal SIGCONT peut éventuellement être envoyé s'il y avait des processus stoppés lorsque qu'il y a la dissociation du terminal de contrôle.
Pour mieux comprendre l'envoi du signal SIGHUP et dans quels circonstances celui-ci est envoyé, le programme ci-dessous demontre le concept :

Listing 4 – sighup.c

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

static void handleSigHup(int sig) {
    printf("%ld: J'ai recu SIGHUP\n", (long) getpid());
    fflush(stdout);
}

int main(int argc, char *argv[]) {

    pid_t processFils;

    struct sigaction sa;
    sa.sa_handler = handleSigHup;

    // Handler pour SIGHUP
    if (sigaction(SIGHUP, &sa, NULL) == -1)
        perror("Erreur sigaction");
        fflush(stdout);

    processFils = fork();

    if(processFils > 0) {
        printf("Je suis le processus parent : PID=%ld; PPID=%ld;
            PGID=%ld; SID=%ld\n", (long) getpid(),
```

```
        (long) getppid(), (long) getpgrp(), (long) getsid(0));
        fflush(stdout);
    }

    if(processFils < 0)
        perror("Erreur fork du fils");
        fflush(stdout);

    if(processFils == 0){
        /*
         * Deplacer le fils dans un nouveau groupe si un parametre a ete
             passe
         * (le shell n'a pas creee ce groupe, il n'est pas dans la liste
             de ses jobs !)
         */
        if(argc > 1) {
            if (setpgid(0, 0) == -1)
                perror("setpgid");
            printf("Le fils a change de groupe !\n");
            fflush(stdout);
        }
        printf("Je suis le processus fils : PID=%ld; PPID=%ld; PGID=%ld;
            SID=%ld\n", (long) getpid(),
        (long) getppid(), (long) getpgrp(), (long) getsid(0));
        fflush(stdout);
    }

    alarm(30); // Envoyer un SIGALARM, pour terminer le process si rien
        ne le termine


    for(;;) { // Wait des signaux
        pause();
    }

}
```

7.1 Cas particuliers

7.1.1 *nohup*

Une commande particulière qui permet d'ignorer le signal SIGHUP, est la commande *nohup*. Cette commande permet d'exécuter un processus/job et ignorer SIGHUP, même lorsque le processus de contrôle ou le terminal de contrôle se termine (comme expliqué précédemment ces deux évènements génèrent le signal SIGHUP). Tout output généré par le processus/job sera redirigé vers nohup.out. Et bien sûr si le processus/job essaye de lire depuis le terminal de contrôle alors que celui-ci n'existe plus, une erreur sera retourné.

8 Références

— The Design and Implementation of the FreeBSD Operating System, by Marshall Kirk McKusick, George V. Neville-Neil
— The Linux Programming Interface by Michael KerrisK
— Advanced Programming in the UNIX Environment : Second Edition by W. Richard Stevens, Stephen A. Rago
— The GNU C Library (glibc) manual