Rapport SYSG5 : Contrôle 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 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 job !
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 le job, 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 au PID 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);
}
```

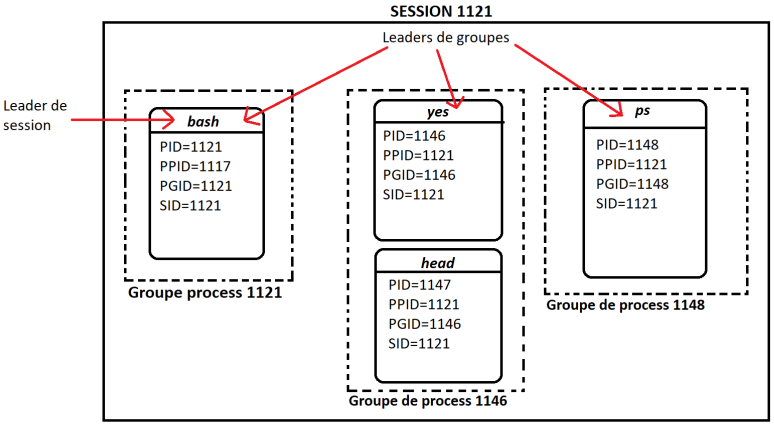
5 Relations Groupes/Sessions

Pour éclaircir les concepts de groupes de processus et de sessions, examinons l'exemple de code shell ci-dessous, offrant une opportunité pratique pour illustrer leur interaction.

```
$ echo $$ # PID du shell
1121
$ yes "data" | head -n 1000000000 > /dev/null & # 2 process en background
[1] 1147
$ 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'
```

Dans cet exemple, le shell a le PID 1121. En lançant les commandes en arrière-plan (*yes* et *head*), deux processus sont créés, et un job est associé à ces deux processus avec le numéro de travail 1. En analysant la sortie de la commande *ps* (qui lui aussi est un job), nous pouvons comprendre la relation des processus.
Le PID 1121 (*bash*) est le shell parent, et les processus 1146 (*yes*), 1147 (*head*), et 1148 (*ps*) ont 1121 comme PPID, indiquant qu'ils sont les enfants du shell. On observe que tous les processus créés ne font pas partie du même groupe que le shell. Cependant, les PID 1146 (*yes*) et 1147 (*head*) ont le même PGID, 1146, montrant qu'ils appartiennent au même groupe de processus, dont le leader est *yes* (PID=PGID). Le SID de tous les processus est 1121, indiquant qu'ils appartiennent tous à la même session.
En résumé, le shell crée une session, et chaque commande lancée forme un groupe de processus au sein de cette session. Les jobs sont associés à des groupes de processus et peuvent avoir plusieurs processus associés.

Voici un schéma pour encore illustrer la relation de ces processus :


SESSION 1121
Leaders de groupes
Leader de session
bash
PID=1121
PPID=1117
PGID=1121
SID=1121
Groupe process 1121
yes
PID=1146
PPID=1121
PGID=1146
SID=1121
head
PID=1147
PPID=1121
PGID=1146
SID=1121
Groupe de process 1146
ps
PID=1148
PPID=1121
PGID=1148
SID=1121
Groupe de process 1148

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 Processus en Background et Foreground

Le terminal de contrôle conserve la notion de groupe de processus en foreground (avant-plan). Au sein d'une session, un seul processus peut être en foreground à un moment donné, tous les autres groupes de processus de la session sont des groupes de processus en background (arrière-plan). Le groupe de processus en foreground est le seul groupe de processus qui peut lire sur le terminal de contrôle. Lorsqu'un des caractères du terminal générateur de signal est tapé sur le terminal de contrôle, le pilote du terminal délivre le signal correspondant aux membres du groupe de processus d'en foreground.

7.1 L'accès au terminal

Comme expliqué précédemment, le groupe de processus en foreground est le seul groupe de processus qui peut lire sur le terminal de contrôle, cette restriction prévient de la concurrence pour l'input du terminal. Si un processus en background essaye de lire depuis le terminal, celui-ci se verra renvoyer un signal SIGINT, dont le comportement par défaut est de stopper le processus/job. Mais il est important de savoir que par défaut les groupes de processus en background peuvent écrire sur le terminal.
Pour illustrer ce comportement utilisions le programme *terminalAccess*. C'est un simple programme qui fait la meme chose que un *cat*, c'est à dire qu'il va lire depuis le terminal ce que l'utilisateur tape aprés avoir entré "Enter". Il gére aussi le signal SIGTTIN.

Listing 4 – terminalAccess.c

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

static void handleSIGTTIN(int sig) {
    printf("%ld: J'ai recu SIGTTIN\n", (long) getpid());
}
int main(void) {
   struct sigaction sa;
   sa.sa_handler = handleSIGTTIN;
   printf("Je suis le processus: PID=%ld; PPID=%ld; PGID=%ld;
       SID=%ld\n", (long) getpid(),
                  (long) getppid(), (long) getpgrp(), (long) getsid(0));

   // Handler pour SIGTTIN
    if (sigaction(SIGTTIN, &sa, NULL) == -1) {
      perror("Erreur sigaction SIGINT");
      exit(-1);
   }
```

```
   char buffer[1024];
   long n;

   printf("Je vais lire depuis le terminal...\n");
   printf("Tapez quelque chose puis Enter, j'afficherais ce que vous
       avez tape...\n");
   while(1) {
      n = read(STDIN_FILENO, buffer, sizeof(buffer));
      if(n < 0) {
         perror("Erreur dans read");
         break;
      }

      write(STDOUT_FILENO, buffer, n);
   }
   return 0;
}
```

Si nous l'éxécutons en foreground rien de particulier se passe, le programme marche correctement. Par contre si celui-ci s'éxécute en background, etant donné que le processus sera en background il n'aura plus accés en *stdin* au terminal, le signal SIGTTIN sera retourné. Si le signal est geré ou ignoré une erreur sera renvoyé lors de la tentative de lecture du terminal (c'est le cas dans ce programme de démonstration).

```
$ ./terminalAccess &
[1] 4623
Je suis le processus: PID=4623; PPID=4595; PGID=4623; SID=4595
Je vais lire depuis le terminal...
Tapez quelque chose puis Enter, j'afficherais ce que vous avez tape...
4623: J'ai recu SIGTTIN
Erreur dans read: Interrupted system call
[1]+ Fini                 ./terminalAccess
```

Un petit cas particulier à mettre en avant est qu'il est aussi possible d'envoyer un signal pour "interdire" à un processus en arrière-plan d'écrire dans le terminal. Si le flag TOSTOP est levé pour le terminal, toute tentative d'un processus en arrière-plan d'écrire se verra renvoyer un signal SIGTTOU (par défaut, ce signal stoppe le processus). Si le processus ignore ou gère ce signal, il pourra quand même écrire sur le terminal.

```
$ stty tostop
$ ./terminalAccess &
[1] 4623
Je suis le processus: PID=4623; PPID=4595; PGID=4623; SID=4595
Je vais lire depuis le terminal...
Tapez quelque chose puis Enter, j'afficherais ce que vous avez tape...
4623: J'ai recu SIGTTIN
```

```
Erreur dans read: Interrupted system call
[1]+ Fini                 ./terminalAccess
```

7.2 Fonctions `tcgetpgrp()` et `tcsetpgrp()`

Les fonctions `tcgetpgrp()` et `tcsetpgrp()` jouent un rôle important dans la manipulation des groupes de processus associés à un terminal, notamment dans les shells de contrôle de jobs. Explorons ces fonctionsl :

- `pid_t tcgetpgrp(int fd)` : La fonction `tcgetpgrp()` est utilisée pour récupérer le groupe de processus en avant-plan (foreground) associé à un terminal spécifique. En tant qu'argument, elle prend le descripteur de fichier `fd` qui est lié au terminal en question. La fonction retourne le PGID du groupe de processus en foreground associé au terminal.
- `int tcsetpgrp(int fd, pid_t pgid)` : La fonction `tcsetpgrp()` est utilisée pour définir le groupe de processus en foreground associé à un terminal spécifique. Elle prend en argument le descripteur de fichier `fd` associé au terminal et le PGID du groupe de processus que l'on souhaite définir en foreground. Le retour de la fonction indique le succès ou l'échec de l'opération.

Remarques :

- Sur de nombreuses implémentations Unix, y compris Linux, ces deux fonctions sont souvent implémentées en utilisant des opérations ioctl (Entrées/Sorties Contrôlées) non standardisées, telles que `TIOCGPGRP` et `TIOCSPGRP`.
- Ces fonctions sont particulièrement utiles dans le contexte des shells de contrôle de jobs, permettant aux shells de gérer les processus en foreground et en background associés à un terminal.

Une démonstration illustrant une de ces fonctions sera vu plus tard (Contrôle de jobs).

8 Le signal SIGHUP

Lorsqu'une session, représentée par le processus contrôleur, perd sa connexion avec le terminal de contrôle (par exemple, lorsque la fenêtre du terminal est fermée), tous les processus de cette session perdent leur association avec le terminal. Pour informer ces processus de cet événement, le noyau envoie un signal, SIGHUP, au processus contrôleur. Ce dernier déclenche ensuite une réaction en chaîne en transmettant le signal SIGHUP à tous les groupes de processus qu'il a créés. Il est important de noter que ce signal n'est pas envoyé aux groupes de processus (jobs) qui n'ont pas été créés par le processus contrôleur.
Par défaut, le signal SIGHUP provoque la terminaison d'un programme. Cependant, un programme peut créer un gestionnaire (handler) pour ce signal, lui permettant de continuer son exécution et d'effectuer des opérations de nettoyage, par exemple. Si des processus étaient arrêtés lorsque la dissociation du terminal de contrôle a eu lieu, le signal SIGCONT peut éventuellement être envoyé pour les réactiver.
Pour une meilleure compréhension du signal SIGHUP et des circonstances dans lesquelles il est déclenché, le programme ci-dessous, *sighup.c*, illustre le concept. Le programme vise à créer un processus enfant, mettre en pause le parent et l'enfant pour capturer le signal SIGHUP. Si un argument facultatif est fourni en ligne de commande, l'enfant rejoint un nouveau groupe de processus dans la même session. Le but est de démontrer que le shell ne transmet pas SIGHUP à un groupe de processus non créé par lui.

Listing 5 – sighup.c

```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();
    }
}
```

Pour utiliser ce programme, vous pouvez exécuter deux instances avec l'une sans arguments et l'autre avec un argument (le processus fils se déplacera dans un nouveau groupe). Chaque instance aura son flux de sortie redirigé vers un fichier log différent pour conserver une trace des affichages. Ensuite, fermez la fenêtre du terminal.

```
$ echo $$ # PID du shell
5533
$ ./sighup > logsAlpha.log 2>&1 &
$ ./sighup a > logsBeta.log 2>&1 &
```

Si vous examinez les logs du premier programme, le fichier contiendra des indications que les deux processus créés ont bien reçu SIGHUP.

```
$ cat logsAlpha.log
Je suis le processus fils : PID=5612; PPID=5611; PGID=5611; SID=5533
```

```
Je suis le processus parent : PID=5611; PPID=5533; PGID=5611; SID=5533
5611: J'ai recu SIGHUP
5612: J'ai recu SIGHUP
```

Si vous examinez les logs du deuxième programme, le fichier contiendra des indications que seulement le parent a bien reçu SIGHUP, l'enfant étant déplacé dans un autre groupe non créé par le shell, celui-ci ne reçoit pas de SIGHUP.

```
$ cat logsBeta.log
Je suis le processus fils : PID=5614; PPID=5613; PGID=5614; SID=5533
Je suis le processus parent : PID=5613; PPID=5533; PGID=5613; SID=5533
5613: J'ai recu SIGHUP
```

8.1 Cas particuliers

8.1.1 Groupes de processus orphelins

Le signal SIGHUP n'est pas envoyé exclusivement quand il y a une perte d'accès au terminal de contrôle. Ce signal peut également envoyé par le noyau dans une situation assez particulière appelé groupes de processus orphelins. Comme le nom l'indique ce sont des groupes de processus qui n'ont plus de parent existant. Ce genre de cas arrive lorsqu'un leader de groupe de processus se termine avant ses membres de ce groupe.
C'est un cas important à gérer puisque si le processus leader de ce groupe se termine, le processus contrôleur (le shell) retire ce groupe de sa liste de jobs, ce qui veut dire que plus aucun processus veille sur les membres de ce groupe ! C'est assez problématique puisque si par hasard des membres de ce groupe de processus orphelin était stoppé, il n'y a plus aucun moyen de le remettre à s'exécuter. Même l'adoption par init n'aiderait pas dans ce cas, puisqu'il gère que les zombies et non les processus stoppés.
Pour prévenir à ce problème le noyau refait usage de SIGHUP. Lorsqu'un groupe de processus devient orphelin le noyau envoie SIGHUP à tous les membres de ce groupe pour les informer des leurs situation, un signal SIGCONT peut être aussi envoyé si un des membres de ce groupe était stoppé, pour s'assurer qu'il reprenne son exécution.

Pour une meilleure compréhension de ce cas particulier et les circonstances dans lesquelles il est déclenché, le programme ci-dessous, *sighupOrphelins.c*, illustre le concept. Le programme mets en place des gestionnaires de signaux pour SIGHUP et SIGCONT. Le programme crée ensuite deux processus fils, chaque fils se stoppe juste aprés sa création.
Une fois que tous les processus fils sont créés, le processus parent attend quelques secondes pour permettre aux fils de s'initialiser. Il quitte ensuite, à ce moment le groupe de processus contenant les fils devient donc orphelin.
Une éxécution de ce programme entraine un résultat similaire à celui-ci :

```
$ ./sighupOrphelins
Je suis le process parent: PID=1005; PPID=942; PGID=1005: SID=942
Je suis un process fils, je vais me stopper: PID=1007; PPID=1005;
    PGID=1005: SID=942
Je suis un process fils, je vais me stopper: PID=1006; PPID=1005;
    PGID=1005: SID=942
Je suis le parent PID=1005; mes fils vont devenir un groupe orphelin...
1007 : J'ai recu SIGCONT
1007 : J'ai recu SIGHUP
1006 : J'ai recu SIGCONT
1006 : J'ai recu SIGHUP
```

Listing 6 – sighupOrphelins.c

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

static void handleSigHupSigCont(int sig) {
    if(sig == 1) {
        printf("%ld: J'ai recu SIGHUP\n", (long) getpid());
    } else {
        printf("%ld: J'ai recu SIGCONT\n", (long) getpid());
    }
}

int main(void) {

    pid_t processFils;
    struct sigaction sa;
    sa.sa_handler = handleSigHupSigCont;

    printf("Je suis le processus parent : PID=%ld; PPID=%ld; PGID=%ld;
        SID=%ld\n", (long) getpid(),
                (long) getppid(), (long) getpgrp(), (long) getsid(0));

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

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

    for(int i = 0; i < 2; i++) {
        switch (fork()) {
            case -1:
```

```
            perror("Erreur fork");
        case 0:
            printf("Je suis un processus fils et je vais me stopper :
                PID=%ld; PPID=%ld; PGID=%ld; SID=%ld\n", (long)
                getpid(),
                (long) getppid(), (long) getpgrp(), (long) getsid(0));

            raise(SIGSTOP); // Stopper fils
            exit(0);
        default:
            break;
    }
}

sleep(3); // Sleep parent pour donner chance aux fils de s'exec
printf("Je suis le parent PID=%ld; mes fils vont devenir un groupe
    de process orphelins....\n", (long) getpid());
exit(0);
}
```

8.1.2 *nohup*

La commande spéciale *nohup* offre la possibilité d'ignorer le signal SIGHUP lors de l'exécution d'un processus ou d'un job. Cette commande permet au processus de continuer son exécution même si le processus de contrôle ou le terminal de contrôle se termine. Comme mentionné précédemment, la fin de ces deux événements génère le signal SIGHUP. Toute sortie générée par le processus sera redirigée vers le fichier *nohup.out*. Cependant, si le processus tente de lire depuis le terminal de contrôle après sa disparition, une erreur sera signalée.
Pour illustrer l'utilisation de *nohup*, considérons un petit script nommé *scriptBidon.sh*. Ce script affiche simplement la date et l'heure toutes les 3 secondes. En le lançant en arrière-plan, puis en affichant la liste des jobs du shell, on peut observer qu'il continue à s'exécuter. La commande *ps.* nous montre également que le processus est attaché à un terminal nommé `pts/0`.

```
$ nohup ./scriptBidon.sh &
[1] 5725
nohup: ignoring input and appending output to 'nohup.out'
$ jobs -l
[1]  + 5725 running  nohup ./scriptBidon.sh
$ ps -ef | grep '[s]criptBidon'
user0  5725  5656  0 18:46 pts/0   00:00:00 /bin/bash ./scriptBidon.sh
```

En fermant ensuite le terminal actuel et en ouvrant un nouveau terminal, les commandes suivantes nous montrent que le script continue son exécution, ayant ignoré le signal SIGHUP, mais n'est plus attaché à un terminal de contrôle (le " ?" dans la colonne TTY indique que le processus n'est pas attaché à un

terminal de contrôle).

```
$ ps -ef | grep '[s]criptBidon'
kent   5725      1  0 18:59 ?    00:00:00 /bin/bash ./scriptBidon.sh
```

8.1.3 *disown*

Une autre commande intéressante pour "immuniser" un processus contre la réception du signal SIGHUP est *disown*. Cette commande permet de retirer un processus de la liste des jobs du shell, tout en le laissant s'exécuter en arrière-plan.
Prenons pour exemple le script *scriptBidon.sh*. Lorsqu'on le lance en arrière-plan et qu'on affiche la liste des jobs du shell, on constate qu'il est toujours présent. Si, par la suite, nous utilisons *disown* et réaffichons la liste des jobs, on observe que le processus a disparu, bien qu'il continue de s'exécuter. Ceci est confirmé en utilisant la commande *ps*.

```
$ ./scriptBidon.sh > blob.log &
[1] 4595
$ jobs -l
[1] + 4595 running ./scriptBidon.sh > blob.log &
$ disown
$ jobs -l
$ ps -ef | grep '[s]criptBidon'
user0 4595 4352 0 11:03 pts/3 00:00:00 /bin/bash ./scriptBidon.sh >
    blob.log &
```

En fermant le terminal actuel et en ouvrant un nouveau terminal, les commandes suivantes révèlent que le script continue son exécution, ayant ignoré le signal SIGHUP. Il n'est cependant plus attaché à un terminal de contrôle (le " ?" dans la colonne TTY indique que le processus n'est pas attaché à un terminal).

```
$ ps -ef | grep '[s]criptBidon'
user0 4595 1 0 11:03 ? 00:00:00 /bin/bash ../scriptBidon.sh > blob.log &
```

9 Contrôle de jobs dans le shell

La section suivante explore les commandes et l'utilisation du contrôle de jobs dans le shell.

Lorsqu'une commande est suivie d'une esperluette (&), elle s'exécute en background (arrière-plan). Chaque travail en arrière-plan est identifié par un numéro unique attribué par le shell. Par exemple :

```
$ sleep 50 &
[1] 112 # Job 1 : process cat avec PID 112
$ sleep 55 &
[2] 114 # Job 2 : process ls avec PID 114
```

Le numéro qui suit le numéro de travail est le PID processus créé pour exécuter la commande, ou, dans le cas d'un pipeline, le PID du dernier processus du pipeline. Afin de faire référence à un job particulier, la notation `%num` est nécessaire, où "num" est le numéro de job assigné par le shell.

La commande intégrée du shell `jobs` liste tous les jobs en arrière-plan :

```
$ jobs
[1]- Running sleep 50 &
[2]+ Running sleep 55 &
```

À ce stade, le shell est le processus en avant-plan pour le terminal. Comme seul un processus en avant-plan peut lire l'entrée du terminal et recevoir des signaux générés par le terminal, il est parfois nécessaire de déplacer un travail en arrière-plan vers l'avant-plan. Cela se fait en utilisant la commande intégrée du shell `fg` :

```
$ jobs
$ fg %1
sleep 50 &
```

Comme le montre cet exemple, le shell réaffiche la ligne de commande d'un travail chaque fois que le travail est déplacé entre l'avant-plan et l'arrière-plan. Ci-dessous, nous verrons que le shell le fait également chaque fois que l'état du travail change en arrière-plan.
Lorsqu'un travail s'exécute en avant-plan, nous pouvons le suspendre en utilisant le caractère de suspension du terminal (généralement Control-Z), qui envoie le signal SIGTSTP au groupe de processus en avant-plan du terminal :

```
Tapez Control-Z
[1]+ Stopped sleep 50 &
```

Après avoir tapé Control-Z, le shell affiche la commande qui a été arrêtée en

arrière-plan. Si nécessaire, nous pouvons utiliser la commande `fg` pour reprendre le travail en avant-plan, ou utiliser la commande `bg` pour le reprendre en arrière-plan. Dans les deux cas, le shell reprend le travail arrêté en lui envoyant un signal SIGCONT :

```
$ bg %1
[1]+ sleep 50 &
```

Nous pouvons arrêter un travail en arrière-plan en lui envoyant un signal SIGSTOP :

```
$ kill -STOP %1
[1]+ Stopped sleep 50 &
$ jobs
[1]+ Stopped sleep 50 &
[2]- Running sleep 55 &
$ bg %1   #  Redemarrer le job en arriere-plan
[1]+ sleep 50 &
```

9.1 Démonstration du contrôle de jobs

Afin de mieux comprendre le comportement et l'organisation du shell en ce qui concerne les commandes de contrôle de jobs, le programme ci-dessous permettra de mieux saisir ces notions.
Ce programme de démonstration vise à reproduire le comportement des jobs gérés par un shell, mettant particulièrement en lumière la manière dont les processus dans un pipeline peuvent être organisés en un job. Cette démonstration est conçue pour illustrer le comportement de plusieurs instances s'exécutant dans un pipeline, comme illustré ci-dessous :

```
$ ./job | ./job
```

Listing 7 – jobControl.c

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

void sigint_handler(int signo) {
   fprintf(stderr,"PID: %d: J'ai recu SIGINT\n", getpid());
}

void sigtstp_handler(int signo) {
   fprintf(stderr,"PID: %d: J'ai recu SIGTSTP\n",
    getpid(), getppid(), getpgrp(), getsid(0));
```

```
   raise(SIGSTOP); // Suspendre le process
}

void sigcont_handler(int signo) {
   fprintf(stderr,"PID: %d: J'ai recu SIGCONT\n",
    getpid(), getppid(), getpgrp(), getsid(0));
   if (isatty(STDIN_FILENO) && (getpid() == getpgrp())) {
    // Print groupe foreground ssi stdin est un terminal et que
    // le process est un leader de groupe
      fprintf(stderr,"Groupe en foreground: %d\n",
          tcgetpgrp(STDIN_FILENO));
   }
}

int main() {
   // Handlers pour SIGINT, SIGTSTP et SIGCONT
   signal(SIGINT, sigint_handler);
   signal(SIGTSTP, sigtstp_handler);
   signal(SIGCONT, sigcont_handler);

    if (isatty(STDIN_FILENO) && (getpid() == getpgrp())) {
    // Print groupe foreground ssi stdin est un terminal et que
    // le process est un leader de groupe
      fprintf(stderr,"Groupe en foreground: %d\n",
          tcgetpgrp(STDIN_FILENO));
   }

   fprintf(stderr,"Je suis le process: PID: %d, PPID: %d, PGID: %d, SID:
       %d\n",
    getpid(), getppid(), getpgrp(), getsid(0));

   while (1) {
      pause();
   }

   return 0;
}
```

La session de shell suivante démontre l'utilisation du programme.

```
$ echo $$ # PID du shell
2062
$ ./job | ./job &
[1] 4511
Je suis le process: PID: 4511, PPID: 2062, PGID: 4510, SID: 2062
Groupe en foreground: 2062
Je suis le process: PID: 4510, PPID: 2062, PGID: 4510, SID: 2062
```

À partir du résultat ci-dessus, nous pouvons voir que le shell est le processus en foreground (`tcgetpgrp()`) pour le terminal. On remarque aussi que tous les processus sont dans la même session que le shell et dans un groupe différent du shell.
Ramenons maintenant ce job en foreground à l'aide de `fg`. Et tapons CTRL-C pour envoyer un signal SIGINT

```
$ fg
./job | ./job
# Tapez Control-C pour generer SIGINT
PID: 4511: J'ai recu SIGINT
PID: 4510: J'ai recu SIGINT
```

À partir de la sortie ci-dessus, nous voyons que le signal SIGINT a été délivré à tous les processus dans le groupe en foreground du terminal (à savoir le job avec les programmes). Le programme ne s'arrête pas puisqu'il gère ce signal.
Utilisons maintenant SIGTSTP à l'aide de Control-Z

```
# Tapez Control-Z pour generer SIGTSTP
PID: 4511: J'ai recu SIGTSTP
PID: 4510: J'ai recu SIGTSTP
[1]+ Stopped            ./job | ./job
```

Tous les processus du job sont maintenant à l'arrêt. Le shell devient donc le groupe en foreground et a la main sur le terminal.
Relançons maintenant le job en background à l'aide de `bg`.

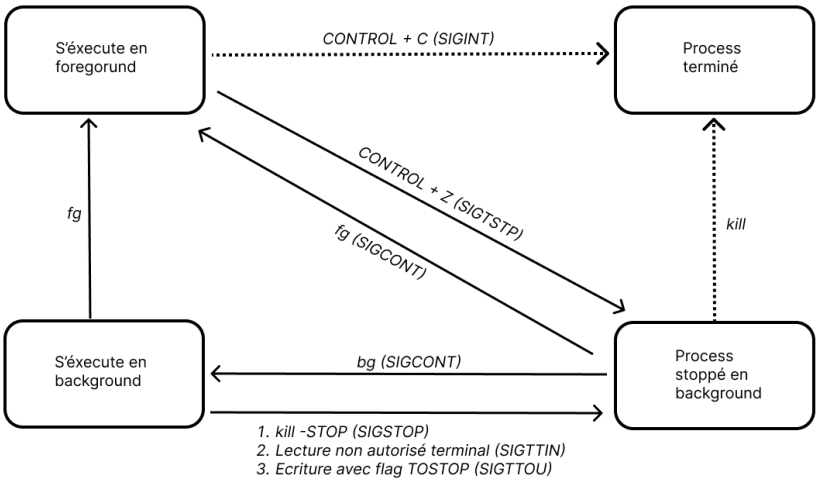
```
$ bg
[1]+ ./job | ./job &
PID: 4511: J'ai recu SIGCONT
PID: 4510: J'ai recu SIGCONT
Groupe en foreground: 2062
```

On remarque donc que tous les processus du job ont été délivrés SIGCONT, et reprennent leurs exécutions.
Pour nettoyer la démonstration, la commande "kill" peut être utilisée.

```
$ kill %1
[1]- Terminated ./job | ./job
```

9.1.1 Schéma récapitulatif

Le schéma ci-dessous illustre les differents états d'un processus sous le fonctionnement de contrôle de jobs.


S'éxecute en foregorund
CONTROL + C (SIGINT)
Process terminé
CONTROL + Z (SIGTSTP)
fg (SIGCONT)
fg
kill
S'éxecute en background
bg (SIGCONT)
Process stoppé en background
1. kill -STOP (SIGSTOP)
2. Lecture non autorisé terminal (SIGTTIN)
3. Ecriture avec flag TOSTOP (SIGTTOU)

10 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