Dev Sys

PRÉSENTÉ PAR : JOSUÉ RATOVONDRAHONA

Version 2021

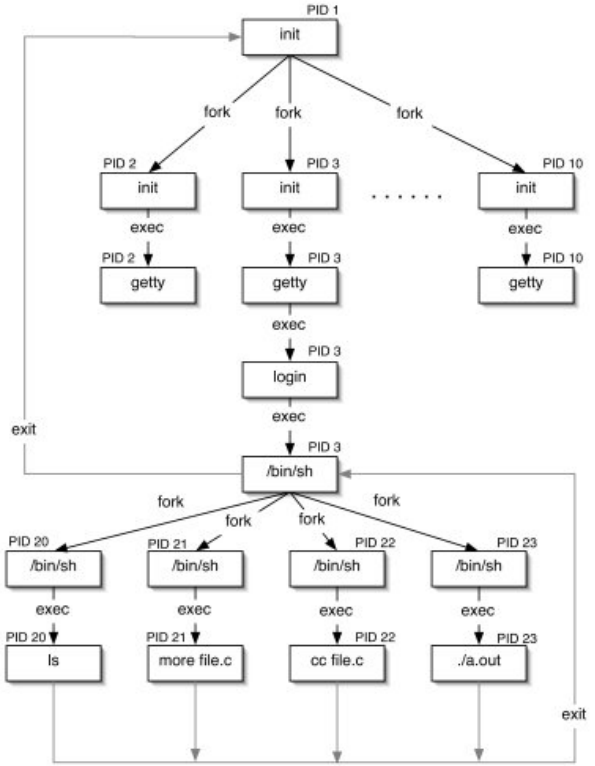
SOMMAIRE

PROCESSUS

1. **Sous Unix, toute tâche qui est en cours d'exécution est représentée par un processus.**
2. **Un processus est une entité comportant à la fois des données et du code.**
3. **Commande Unix : ps -aux**

PROCESSUS

1. FORK
2. PID
3. UID
4. GID
5. Groupe de Processus
6. Session
7. Capacité d’un Processus



PROCESSUS

1. **Rappel: système de fichier et privilège LINUX**

PROCESSUS

FORK

La fonction fork permet à un programme en cours d'exécution de créer un nouveau processus. Le processus d'origine est appelé processus père, et il garde son **PID**, et le nouveau processus créé s'appelle processus fils, et possède un nouveau PID. Le processus père et le processus fils ont le même code source, mais la valeur retournée par fork permet de savoir si on est dans le processus père ou fils. Ceci permet de faire deux choses différentes dans le processus père et dans le processus fils (en utilisant un if et un else ou un switch), même si les deux processus on le même code source.

La fonction fork retourne -1 en cas d'erreur, retourne 0 dans le processus fils, et retourne le PID du fils dans le processus père. Ceci permet au père de connaître le PID de son fils.

PROCESSUS

FORK

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
    int main (void)
{
    pid_t pid_fils;
    pid_fils = fork();
    if (pid_fils == -1)
    {
        puts("Erreur de creation du nouveau
processus");
        exit (1);
    }
    if (pid_fils == 0)
    {
        printf("Nous sommes dans le fils\n");
        /* la fonction getpid permet de connaître son
 propre PID */
        printf("Le PID du fils est \%d\n", getpid());
        /* la fonction getppid permet de connaître le
 PPID(PID de son père) */
        printf("Le PID de mon père (PPID) est \%d",
getppid());
    }
    else
    {
        printf("Nous sommes dans le père\n");
        printf("Le PID du fils est \%d\n", pid_fils);
        printf("Le PID du père est \%d\n", getpid());
        printf("PID du grand-père : \%d", getppid());
    }
return 0;
}
```

PROCESSUS

PID

1. Identification par le PID

Le premier processus du système, init, est créé directement par le noyau au démarrage.

La seule manière, ensuite, de créer un nouveau processus est d'appeler l'appel-système fork( ), qui va dupliquer le processus appelant. Au retour de cet appel-système, deux processus identiques continueront d'exécuter le code à la suite de fork( ).

L'appel-système fork( ) est déclaré dans <unistd.h>, ainsi :

```
pid_t fork(void);
```

PROCESSUS

UID

2. Identification de l'utilisateur correspondant au processus **UID**

un système Unix est particulièrement orienté vers l'identification de ses utilisateurs. Toute activité entreprise par un utilisateur est soumise à des contrôles stricts quant aux permissions qui lui sont attribuées. Pour cela, chaque processus s'exécute sous une identité précise. Dans la plupart des cas, il s'agit de l'identité de l'utilisateur qui a invoqué le processus et qui est définie par une valeur numérique : **l'UID (User IDentifier).**

PROCESSUS

UID

2. Identification de l'utilisateur correspondant au processus **UID**

Il existe trois identifiants utilisateurs: **UID réel**, **UID effectif**, **UID sauvé**

Réel : lance le programme

Effectif : correspond aux privilèges accordés au processus

Sauvé : est une copie de l'ancien UID effectif lorsque celui-ci est modifié.

PROCESSUS

UID

2. Identification de l'utilisateur correspondant au processus **UID**

Les appels-systèmes : réel et effectif

```
uid_t getuid(void);
uid_t geteuid(void);
```

PROCESSUS

GID

3. Identification du groupe d'utilisateur correspondant au processus **GID**

Chaque utilisateur du système appartient à un ou plusieurs groupes. Ces derniers sont définis dans le fichier **/etc/groups** . Un processus fait donc également partie des groupes de l'utilisateur qui l'a lancé. Comme nous l'avons vu avec les **UID**, un processus dispose donc de plusieurs **GID (Group IDentifier) réel, effectif, sauvé**, ainsi que de GID supplémentaires si l'utilisateur qui a lancé le processus appartient à plusieurs groupes.

PROCESSUS

GID

3. Identification du groupe d'utilisateur correspondant au processus **GID**

Le GID réel correspond au groupe principal de l'utilisateur ayant lancé le programme (celui qui est mentionné dans /etc/passwd). Le GID effectif peut être différent du GID réel si le fichier exécutable dispose de l'attribut Set-GID (chmod g+s). C'est le GID effectif qui est utilisé par le noyau pour vérifier les autorisations d'accès aux fichiers.

La lecture de ces GID se fait symétriquement à celle des UID avec les appels-système getgid() et getegid() . La modification (sous réserve d'avoir les autorisations nécessaires) peut se faire à l'aide des appels setgid(), setegid() et setregid() .

PROCESSUS

GID

3. Identification du groupe d'utilisateur correspondant au processus **GID**

```
gid_t getgid(void);
gid_t getegid(void);
int setgid(gid_t egid);
int setegid(gid_t egid);
int setregid(gid_t rgid, gid_t egid);
```

Les deux premières fonctions renvoient le GID demandé, les deux dernières renvoient 0 si elle réussissent et –1 en cas d'échec.

PROCESSUS

Groupe de processus

4. Identification de groupe de processus

Les processus sont organisés en groupes. Rappelons qu'il ne faut pas confondre les groupes de processus avec les groupes d'utilisateurs que nous venons de voir, auxquels appartiennent les processus. Les groupes de processus permettent l'envoi global de signaux à un ensemble de processus. Ce concept, tout comme l'identificateur de session que nous verrons immédiatement à la suite, sert surtout aux interpréteurs de commandes – les shells – pour implémenter le contrôle des jobs. La prise en considération des groupes de processus dans les applications classiques est rare.

```
pid_t getpgid (pid_t pid);
```

PROCESSUS

SESSION

5. Identification de session

Il existe finalement un dernier regroupement de processus, **les sessions**, qui réunissent divers groupes de processus. Les sessions sont très **liées à la notion de terminal de Contrôle des processus**. Il n'y a guère que les **shells** ou **les gestionnaires de fenêtres pour les environnements graphiques** qui ont besoin de gérer les sessions. Une exception toutefois : les applications qui s'exécutent sous forme de **daemon** doivent accomplir quelques formalités concernant leur session.

PROCESSUS

SESSION

5. Identification de session

Généralement, une **session** est attachée à un terminal de contrôle, celui qui a servi à la connexion de l'utilisateur. Avec l'évolution des systèmes, les terminaux de contrôle sont souvent **des pseudo-terminaux virtuels gérés par les systèmes graphiques de fenêtrage ou par les pilotes de connexion réseau**. Au sein d'une session, **un groupe de processus est en avant-plan** ; il reçoit directement les données saisies sur le clavier du terminal, et peut afficher ses informations de sortie sur l'écran de celui-ci. **Les autres groupes de processus de la session s'exécutent en arrière-plan**.

PROCESSUS

SESSION

5. Identification de session

Pour **créer une nouvelle session**, un processus **ne doit pas être leader de son groupe**. En effet, la création de la session passe par **une étape de constitution d'un nouveau groupe de processus prenant l'identifiant du processus appelant**. Il est indispensable que **cet identifiant ne soit pas encore attribué à un groupe qui pourrait contenir éventuellement d'autres processus.**

```
pid_t setsid (void);
```

PROCESSUS

Capacité de Processus

6. Capacité de processus

Depuis Linux 2.2, **la toute-puissance d'un processus exécuté sous l'UID effectif root peut être limitée**. Une application dispose à présent d'un jeu de capacités permettant de définir ce que le processus peut faire sur le système. **Cela est défini dans le document Posix.1e (anciennement Posix.6).**

PROCESSUS

Capacité de Processus

6. Capacité de processus

Les capacités d'un processus correspondent à des privilèges, aussi les applications courantes ont-elles des ensembles de capacités vides. En dotant un programme d'un jeu restreint de privilèges; par exemple pour modifier sa propre priorité d'ordonnancement, on lui accorde/une puissance suffisante pour accomplir son travail, tout en évitant tout problème de sécurité qui pourrait survenir si le programme était détourné de son utilisation normale. Ainsi, même si une faille de sécurité existe dans l'application, et si elle est découverte par un utilisateur malintentionné, celui-ci ne pourra exploiter que le privilège accordé au programme et pas d'autres capacités dangereuses réservées habituellement à root (par exemple pour insérer un module personnel dans le noyau).

PROCESSUS

Capacité de Processus

6. Capacité de processus

Un processus dispose de trois ensembles de capacités :

- L’ensemble des capacités effectives
- L’ensemble des capacités transmissibles
- L’ensemble des capacités possibles

PROCESSUS

Capacité de Processus

6. Capacité de processus

Une application a le droit de réaliser les opérations suivantes sur ses capacités :

- On peut mettre dans l'ensemble effectif ou l'ensemble transmissible n'importe quelle capacité.
- On peut supprimer une capacité de n'importe quel ensemble.

EXERCICES

- 3.1- Ecrire un programme qui crée **un fils**. Le père doit afficher “je suis le père” et le fils doit afficher “je suis le fils”.
- 3.2- Ecrire un programme qui crée **deux fils appelés fils 1 et fils 2.** Le père doit afficher “je suis le père” et le fils 1 doit afficher “je suis le fils 1”, et le fils 2 doit afficher “je suis le fils 2”
- 3.3- Ecrire un programme qui crée **deux fils appelés fils 1 et fils 2**. Chaque fils doit attendre un nombre de secondes aléatoire entre 1 et 10, en utilisant la fonction **sleep**. Le programme attend que le fils le plus long se termine et affiche la durée totale. On pourra utiliser la fonction time de la bibliothèque **time.h**, qui retourne le nombre de secondes depuis le premier janvier 1970 à 0h (en temps universel)

PROCESSUS

RAPPEL

Commande PS permet de donner la liste des processus :

```
% ps
PID   TTY   TIME     CMD
21693 pts/8 00:00:00 bash
21694 pts/8 00:00:00 ps
```

PROCESSUS

RAPPEL

PROCESSUS IDENTIFIER PAR:

PID : Processus identifiant

UID : Utilisateur identifiant

GID : Groupe d'utilisateur identifiant

Groupe de processus :

Exemple commande ls (liste un répertoire). Sur l'écran on voit la liste du répertoire mais en arrière plan des processus tourne pour afficher la liste.

Session : sert à contrôler les groupes de processus lancer par un utilisateur.

Capacité de processus : limite des actions de root sur un processus ou groupe de processus.

PROCESSUS

RAPPEL

Fonction FORK

Créer un nouveau processus en créant un ou plusieurs processus fils

APPEL SYSTÈME EXEC

- **Famille Fonction appel système EXEC:**
- EXECL : La fonction execl prend en paramètre une liste des arguments à passer au programme (liste terminée par NULL)
- EXECLP : La fonction execlp permet de rechercher les exécutables dans les répertoires apparaissant dans le PATH, ce qui évite souvent d'avoir à spécifier le chemin complet.
- EXECV : La différence avec execl est que l'on n'a pas besoin de conaître la liste des arguments à l'avance (ni même leur nombre). Cette fonction a pour prototype :

```
int execv(const char* application, const char* argv[]);
```

FONCTION SYSTEM

- **La fonction system** de la bibliothèque stdlib.h permet directement de lancer un programme dans un programme C sans utiliser fork et exec. Pour cela, on utilise l'instruction :

```
#include <stdlib.h>

...

system("commande");
```

EXERCICES SYSTEM-EXEC

- **Ecrire un programme qui crée un processus fils. Le fils affiche la liste d'un répertoire avec la commande execl.**
- **Ecrire le même programme qui crée un processus fils. Le fils affiche la liste d'un répertoire avec la commande execv.**
- **Ecrire un programme qui utilise system en exécutant la commande « clear »**

PROCESSUS

Communication entre Processus

Un tube de communication est un tuyau (en anglais pipe) dans lequel un processus peut écrire des données et un autre processus peut lire. On crée un tube par un appel à la fonction pipe, déclarée dans **unistd.h** :

int pipe(int descripteur[2]);

La fonction renvoie 0 si elle réussit, et elle crée alors un nouveau tube. La fonction pipe remplit le tableau descripteur passé en paramètre, avec :

• **descripteur[0]** désigne la sortie du tube (dans laquelle on peut lire des données) ;

• **descripteur[1]** désigne l'entrée du tube (dans laquelle on peut écrire des données) ;

PROCESSUS

Communication entre Processus

Le principe est **qu'un processus va écrire dans descripteur[1]** et **qu'un autre processus va lire les mêmes données dans descripteur[0].** Le problème est qu'on ne crée le tube dans un seul processus, et un autre processus ne peut pas deviner les valeurs du tableau descripteur.

Pour faire communiquer plusieurs processus entre eux, il faut **appeler la fonction pipe avant d'appeler la fonction fork**. Ensuite, le processus père et le processus fils auront les mêmes descripteurs de tubes, et pourront donc communiquer entre eux. De plus, un tube ne permet de communiquer que dans un seul sens. Si l'on souhaite que les processus communiquent dans les deux sens, **il faut créer deux pipes.**

Pour écrire dans un tube, on utilise la fonction write :

```
ssize_t write(int descripteur1, const void *bloc, size_t taille);
```

PROCESSUS

Communication entre Processus

Le descripteur doit correspondre à l'entrée d'un tube. La taille est le nombre d'octets qu'on souhaite écrire, et le bloc est un pointeur vers la mémoire contenant ces octets.

Pour lire dans un tube, on utilise la fonction read :

*ssize_t read(int descripteur0, void *bloc, size_t taille);*

Le descripteur doit correspondre à la sortie d'un tube, le bloc pointe vers la mémoire destinée à recevoir les octets, et la taille donne le nombre d'octets qu'on souhaite lire. La fonction renvoie le nombre d'octets effectivement lus. Si cette valeur est inférieure à taille, c'est qu'une erreur s'est produite en cours de lecture (par exemple la fermeture de l'entrée du tube suite à la terminaison du processus qui écrit).

PROCESSUS

Communication entre Processus

TUBES NOMMES

On peut faire **communiquer deux processus** à travers un tube nommé. **Le tube nommé passe par un fichier sur le disque**. **L'intérêt est que les deux processus n'ont pas besoin d'avoir un lien de parenté**. Pour créer un tube nommé, on utilise la fonction **mkfifo** de la bibliothèque

sys/stat.h.

PROCESSUS

Communication entre Processus

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#define BUFFER_SIZE 256
int main(void)
{
    pid_t pid_fils;
    int tube[2];
    unsigned char bufferR[256], bufferW[256];
    puts("Cration d'un tube");
    if (pipe(tube) != 0) {/* pipe */
    {
        fprintf(stderr, "Erreur dans pipe\(\backslash\)n");
        exit(1);
    }
    pid_fils = fork(); {/* fork */
    if (pid_fils == -1)
    {
        fprintf(stderr, "Erreur dans fork\(\backslash\)n");
        exit(1);
    }
    if (pid_fils == 0) {/* processus fils */
        {
            printf("Fermeture entre dans le fils (pid = %d)\(\backslash\)n",
getpid());
            close(tube[1]);
            read(tube[0], bufferR, BUFFER_SIZE);
            printf("Le fils (%d) a lu : %s\(\backslash\)n", getpid(), bufferR);
        }
    else {/* processus pre */
    {
        printf("Fermeture sortie dans le pre (pid = %d)\(\backslash\)n",
getpid());
        close(tube[0]);
        sprintf(bufferW, "Message du pre (%d) au fils", getpid());
        write(tube[1], bufferW, BUFFER_SIZE);
        wait(NULL);
    }
return 0;
}
```

PROCESSUS

Communication entre Processus

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
int main()
    {
        int fd;
        FILE *fp;
        char *nomfich="/tmp/test.txt"; /* nom du fichier */
        if(mkfifo(nomfich, 0644) != 0) /* création du fichier */
            {
                perror("Problème de création du noeud de tube");
                exit(1);
            }
        fd = open(nomfich, O_WRONLY); /* ouverture en écriture */
        fp=fdopen(fd, "w"); /* ouverture du flot */
        fprintf(fp, "coucou\(\backslash\)n"); /* écriture dans le flot */
        unlink(nomfich); /* fermeture du tube */
        return 0;
    }
```

PROCESSUS

Communication entre Processus

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
int main()
    {
        int fd;
        FILE *fp;
        char *nomfich="/tmp/test.txt", chaine[50];
        fd = open(nomfich, O_RDONLY); /* ouverture du tube */
        fp=fdopen(fd, "r"); /* ouverture du flot */
        fscanf(fp, "%s", chaine); /* lecture dans le flot */
        puts(chaine); /* affichage */
        unlink(nomfich); /* fermeture du flot */
        return 0;
    }
```

PROCESSUS

Exercices

Communication entre Processus

Écrire un programme qui crée deux processus. Le processus père ouvre un fichier texte en lecture. On suppose que le fichier est composé de mots formés de caractères alphabétiques séparés par des espaces. Le processus fils saisit un mot au clavier. Le processus père recherche le mot dans le fichier, et transmet au fils la valeur 1 si le mot est dans le fichier, et 0 sinon. Le fils affiche le résultat.

PROCESSUS

Threads Posix

Un thread (ou fil d'exécution en français) est une partie du code d'un programme (une fonction), qui **se déroule parallèlement à d'autre parties du programme**. Un premier intérêt peut être **d'effectuer un calcul qui dure un peu de temps** (plusieurs secondes, minutes, ou heures) **sans que l'interface soit bloquée** (le programme continue à répondre aux signaux). L'utilisateur peut alors intervenir et interrompre le calcul sans taper un ctrl-C brutal. Un autre intérêt est **d'effectuer un calcul parallèle sur les machines multi-processeur**. Les fonctions liées aux **thread** sont dans la bibliothèque **pthread.h**, et il faut compiler avec la librairie **libpthread.a**

PROCESSUS

Exemples Threads Posix

Le premier exemple crée un thread qui dort un nombre de secondes passé en argument, pendant que le thread principal attend qu'il se termine.

PROCESSUS

Exemples Threads Posix

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
void *ma_fonction_thread(void *arg)
{
    int nbsec = (int)arg;
    printf("Je suis un thread et j'attends %d secondes\(\backslash\)n",
nbsec);
    sleep(nbsec);
    puts("Je suis un thread et je me termine");
    pthread_exit(NULL); {/* termine le thread proprement */
}
    int main(void)
    {
        int ret;
        pthread_t my_thread;
        int nbsec;
        time_t t1;
        srand(time(NULL));
        t1 = time(NULL);
        nbsec = rand()%10; {/* on attend entre 0 et 9 secondes */
        {/* on cre le thread */
            ret = pthread_create(&my_thread, NULL,
            ma_fonction_thread, (void*)nbsec);
            if (ret != 0)
            {
                fprintf(stderr, "Erreur de cration du thread");
                exit (1);
            }
        pthread_join(my_thread, NULL); {/* on attend la fin du thread */
        printf("Dans le main, nbsec = %d\(\backslash\)n", nbsec);
        printf("Duree de l'operation = %d\(\backslash\)n", time(NULL)-t1);
        return 0;
}
```

PROCESSUS

Exemples Threads Posix

Le deuxième exemple crée un thread qui lit une valeur entière et la retourne au main.

PROCESSUS

Exemples Threads Posix

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
void *ma_fonction_thread(void *arg)
{
    int resultat;
    printf("Je suis un thread. Veuillez entrer un
entier\(\backslash\)n");
    scanf("%d", &resultat);
    pthread_exit((void*)resultat); {/* termine le thread proprement */
}
    int main(void)
    {
        int ret;
        pthread_t my_thread;
        {/* on cre le thread */
        ret = pthread_create(&my_thread, NULL,
        ma_fonction_thread, (void*)NULL);
        if (ret != 0)
            {
                fprintf(stderr, "Erreur de cration du thread");
                exit (1);
            }
    pthread_join(my_thread, (void*)&ret); {/* on attend la fin du
thread */
    printf("Dans le main, ret = %d\(\backslash\)n", ret);
    return 0;
}
```