Cours Système n°4: les processus

Éric Gaudefroy

École Hexagone
Cursus Cyberdéfense - M1

1. Le parallélisme

1.1. Introduction du concept

La notion de multi-tâches n'est pas toujours allée de soi en informatique. En effet, du temps des mainframes, le fonctionnement des ordinateurs reposaient sur l'execution séquentielle de jobs. Un tel système n'avait pas gérer d'interactions avec l'utilisateur. Certaines machines étaient dédiées de telles interactions, les consoles, qui permettaient de soumettre un job, puis d'en récupérer les résultats pour les imprimer par exemple. L'avénement de systèmes distribués multi-utilisateurs et multi-tâche a donné naissance des systèmes permettant de gérer en parallèle plusieurs lots d'exécution. Dans le monde des micro-ordinateurs, MS-DOS est un exemple de système mono-utilisateur et monotâche. Certains mécanismes donnaient cependant parfois l'illusion d'un parallèlisme limité : les interruptions matérielles et les programmes résidents.

Aujourd'hui, les sytèmes d'exploitation mettent quasi-systématiquement en jeu des mécanismes autorisant l'exécution en parallèle de plusieurs programmes.

1.2 Les différents niveaux matériels du parallélisme

Il existe plusieurs niveaux auxquels l'exécution en parallèle peut être visible : des machines autonomes reliées en réseau (l'intérieur de clusters) aux ordinateurs contenant une unité de calcul unique mettant en oeuvre le principe du temps partagé, en passant par des ordinateurs multi-CPU, les plus communs aujourd'hui. A chaque niveau correspondent des mécanismes de communication, de stockage et de synchronisation de caches distincts :

- par exemple, un cluster réseau coordonnera ses différents éléments au niveau applicatif (coordination d'applications via des appels RPC[1]) ou reposera sur une sorte de noyau réparti (POSIX, NFS) ;
- il existe de même plusieurs types d'architectures multi-CPU. Certaines reposent sur un processeur multi-coeur, d'autres sur des processeurs distincts partageant une même mémoire ou disposant de mémoires séparées ;
- dans le cas du temps partagé, le temps CPU est en réalité fractionné entre les différents programmes fonctionnant en parallèle, donnant l'illusion d'un parallélisme sur une architecture intrinsèquement séquentielle.

1. *Remote Procedure Call*

Dans le cas de ressources matérielles partagées, le système d'exploitation doit pouvoir synchroniser les accès à ces ressources pour éviter l'apparition d'incohérences. Cette synchronisation ne peut fonctionner que si elle repose sur des dispositifs matériels appropriés, comme le verrouillage d'une zone mémoire commune, afin de prévenir certaines opérations concurrentes.

Dans la suite du ours, on s'intéressera au mode en temps partagé, qui s'applique au départ un CPU unique, mais qui peut s'étendre des systèmes contenant plusieurs unités de calcul.

1.3. Le temps partagé

Chaque CPU ne permet à un instant donné l'exécution que d'un seul lot d'instructions. Dans un système d'exploitation interactif monotâche, l'interpréteur de commandes attend les instructions au clavier, se bloque pour exéuter la commande voulue, puis se replace en écoute une fois la commande terminée. On peut donc distinguer plusieurs programmes chargés en mémoire, mais il n'y a pas de parallélisme d'exécution. Un tel comportement est généralement inacceptable pour au moins deux raisons :

- les performances : si la commande lancée attend un événement particulier (réponse d'un périphérique, entrée clavier), on aurait pu effectuer d'autres actions en attendant que cet événement ait lieu ;
- le confort d'utilisation : de nombreuses applications interagissant avec l'utilisateur ne nécessitent que peu de ressources CPU (éditeur de texte, lecteur de fichiers son). Leur exécution simultanée est donc possible et apporte un certain confort d'utilisation.

Ainsi, pour des raisons d'efficacité et d'interactivité, le système va fractionner le temps CPU entre différents programmes qui s'exécuteront rapidement à tour de rôle, donnant l'illusion d'un parallélisme. Cette répartition du temps d'exécution repose sur des mécanismes matériels :

- la pagination mémoire (cf. cours sur la mémoire) qui permet de partager la RAM de façon transparente entre tâches ;
- les interruptions qui permettent, en réponse certains événements, d'interrompre le flot d'instructions en cours de traitement et d'invoquer une fonction pré-enregistrée du noyau ; les interruptions peuvent être générées par un stimulus physique provenant d'un périphérique, par une exception CPU ou par une requête logicielle ;
- des instructions de changement de contexte CPU (changement de pile, des tables de pages, etc...).

Le fonctionnement d'un système d'exploitation peut alors suivre la stratégie suivante :

- le flot d'instructions exécuté au démarrage de la machine initialise l'état du processeur en fonction des besoins de l'OS et enregistre les fonctions de traitement des interruptions ;
- le noyau gère une structure particulière qui est une liste de tâches en attente ; la création d'une nouvelle tâche consiste allonger cette liste ;
- la fonction enregistrée pour traiter les interruptions d'horloge est couplée à un ordonnanceur, capable de sélectionner une nouvelle tâche dans la liste des tâches en attente et de la substituer la tâche qui était en cours d'exécution (changement de contexte) ;
- lorsque le flot du démarrage a fini son travail et programmé le lancement d'autres tâches, il se met volontairement en sommeil et active l'ordonnanceur.

Ce procédé permet, une fois l'ordonnanceur activé, de donner l'illusion que les tâches en attente s'exécutent parallèlement. On peut alors d'une part étudier le fonctionnement et les politiques appliquées par l'ordonnanceur et de l'autre écrire des programmes sans penser au fait qu'ils s'exécutent

parallèlement à d'autres tâches (aux problèmes de synchronisation et de prévention des *race conditions* près !).

2. L'ordonnancement

2.1. Répartition du temps CPU sur un système standard

Lors du fonctionnement de la machine, le temps CPU est passé :

- dans les tâches utilisateur (root ou non) :

  dans le déroulement du code propre de la tâche et de celui de ses bibliothèques (= flot d'instructions principal),

  dans ses routines de traitement des signaux reçus,

  côté noyau, dans le traitement des appels système (potentiellement non interruptibles par l'ordonnanceur, sauf s'ils sont bloquants),

  côté noyau, dans le traitement des exceptions processeur générées par l'exécution du code du processus, de son fait (ex : division par zéro) ou non (ex : erreur de pagination due au *swap*) ;
- dans les threads noyau, qui sont essentiellement gérés comme les tâches utilisateur (elles sont visibles avec certaines options de **ps**), mais dont le code est dans le noyau et qui participent à son fonctionnement ;
- dans le traitement (court) des interruptions externes (horloge, périphériques, etc...).

Remarque : Le temps passé à traiter les interruptions est généralement injustement pris dans celui alloué à la tâche active qui est interrompue.

Pour cette raison, on considère trois façons de comptabiliser le temps d'exécution d'une tâche (cf. **man setitimer, man times**). Les temps mesurés sont :

- le temps réel écoulé (= temps CPU toutes tâches comprises) ;
- le temps virtuel écoulé (= temps CPU passé dans le code utilisateur de la tâche) ;
- le temps d'activité (= temps CPU passé dans le code utilisateur + temps CPU passé dans le code noyau mais au profit de la tâche).

Appels à l'ordonnanceur

En fonction des politiques, l'ordonnanceur peut être appelés à plusieurs moments du fonctionnement du système :

- suite aux interruptions d'horloge, pour éventuellement changer la tâche active ;
- suite à d'autres événements (traitement d'interruptions physiques, d'appels système) ayant débloqué des tâches prioritaires ;
- en cas de gel, de blocage ou de terminaison de la tâche active, pour sélectionner une nouvelle tâche disponible ;
- à la demande de la tâche active, si elle souhaite relacher le CPU.

2.3. Politiques d'ordonnancement Ordonnanceur coopératif / préemptif

Une politique d'ordonnancement est dite préemptive si une tâche en cours d'exécution est susceptible d'être interrompue par l'ordonnanceur en vue de donner la main une autre tâche en attente (partage systématique du temps CPU ou activation d'une tâche plus prioritaire). L'action des politiques non préemptives se limite au choix de la tâche suivante parmi celles en attente lorsque la tâche active relâche le CPU. Ces politiques supposent donc une bonne coopération entre tâches, la tâche active ayant la possibilité de monopoliser le CPU.

2.3.1. Une politique simple : la file d'attente

La politique la plus simple consiste à gérer une FIFO de tâches en attente. Losqu'une tâche se termine, on passe la suivante. Si une tâche est bloquée ou lâche volontairement le CPU, on la place en fin de queue et on active la tâche suivante. Ce partage est loin d'être optimal, notamment du point de vue des tâches courtes qui peuvent attendre longtemps, une fois lancées, avant d'être exécutées. Cette politique est non préemptive.

2.3.2. Avantage aux tâches rapides

Une variante consiste à privilégier les tâches rapides. Ceci nécessite de connaître à l'avance leur durée (cf. le type de mécanisme offert par l'option RLIMIT_CPU de **setrlimit**). Par ailleurs, les tâches longues peuvent ne jamais être programmées si de nouvelles tâches courtes sont sans arrêt créées (risque de famine). Cette politique est par défaut non préemptive ; en version préemptive, elle autorise le remplacement de la tâche courante dès l'apparition d'une tâche plus courte. On oppose souvent les tâches gourmandes en calcul et les tâches interactives. En effet, les premières correspondent à celles qui utilisent intensivement le temps de calcul, alors que les secondes relâchent généralement rapidement le CPU, en attente d'une opération d'entrée/sortie.

2.3.3. Le tourniquet (*round-robin*)

Une politique préemptive relativement simple est celle de l'algorithme du tourniquet (*round-robin*). On définit un quantum de temps au cours duquel une tâche a le droit de s'exécuter sans être interrompue par l'ordonnanceur. A l'expiration du quantum, c'est au tour de la tâche suivante d'obtenir le CPU pour un nouveau quantum de temps. Les tâches en attente sont gérées par une FIFO. La différence vient du caractère forcé du changement de tâche en fin de quantum. Naturellement, cette fin de quantum ne sera pas atteinte si la tâche relâche volontairement le CPU, se bloque ou se termine avant. Techniquement, le quantum de temps doit être un multiple de la fréquence des interruptions d'horloge.

2.3.4. Le temps partagé universel

Enfin, il existe une politique préemptive plus élaborée dite du « temps partagé universel » ou du « recyclage avec priorité variable » . **Il s'agit de la politique utilisée par défaut sur les systèmes UNIX**. Elle repose sur un algorithme du tourniquet enrichi par la gestion de priorités dynamiques. Ainsi, chaque tâche se voit associer une priorité dynamique qui dépend :

- de sa gentillesse (*nice*) vis-à-vis des autres tâches ;
- du temps CPU déjà consommé et/ou passé à attendre le CPU ;

- optionnellement, cette priorité peut être modulée par des facteurs d'optimisation (on peut ainsi favoriser une tâche interactive ayant de nombreuses entrées/sorties ou encore une tâche qui simplifie le changement de contexte).

L'ordonnanceur choisira alors la tâche dont la priorité dynamique est la plus élevée. Selon les implémentations, les tâches non bloquées dont le *nice* est le plus individualiste seront plus souvent sélectionnées et/ou resteront plus longtemps actives (allongement du quantum de temps). L'apparition ou le réveil d'une tâche de priorité dynamique supérieure à celle de la tâche active peut, selon les cas, provoquer un changement de tâche prématuré.

3. Fonctionnement des processus sous Linux

Les tâches utilisateur manipulées par le système sont des processus (subdivisibles en threads, cf. cours dédié). Ils sont organisés selon une hiérarchie :

- le processus initial, **init**, qui a pour *pid* (*process id*) 1, est à la racine de cette hiérarchie ;
- lorsqu'un processus crée un nouveau processus avec l'appel système **fork**, l'ancien processus est appelé père et le nouveau processus est appelé fils. Ce lien de filiation crée donc une hiérarchie de processus ;
- si un processus qui a des enfants se termine, ceux-ci sont rattachés au processus **init**.

3.1. État des tâches

L'ordonnanceur UNIX/Linux distingue quatre états pour les tâches (processus) à gérer :

- **tâche active** : il s'agit de la tâche en train d'être exécutée (il peut y en avoir plusieurs dans une architecture multi-CPU) ;
- **tâches éligibles** : il s'agit de tâches en cours d'exécution et placées en attente de ressources CPU ; ces tâches sont classées en fonction de critères de priorité ;
- **tâches bloquées** : il s'agit d'une tâche ayant exécuté un appel système bloquant (demande d'une ressource), se trouvant dans un état interruptible ou non, ou ayant reçu un signal particulier ;
- **tâches zombies** : il s'agit de tâches terminées dont l'arrêt et le code de terminaison n'ont pas été acquittés.

Les principales transitions entre états sont :

- **Ø→ éligible** : création la demande de la tâche active ;
- **active → éligible** : expiration du quantum de temps, relâche volontaire du CPU ou réveil d'une tâche prioritaire ;
- **active → bloquée** : réalisation d'un appel système bloquant ou réception d'un signal particulier ;
- **active → zombie** : appel la fonction **exit** ou réception d'un signal correspondant à une erreur fatale ;
- **éligible → active** : sélection de la tâche par l'ordonnanceur ;
- **éligible → bloquée** : réception d'un signal particulier ;
- **éligible ou bloquée → zombie** : réception d'un signal fatal ;

- **bloquée → éligible (ou active)** : obtention de la ressource demandée ou réception d'un signal particulier ;
- **zombie → Ø** : acquittement de l'arrêt par le créateur.

Remarque : Il n'y a pas de passage par l'état zombie si le créateur du processus :

- est déjà en train d'attendre le code de terminaison de la tâche qui vient de se terminer ;
- ignore explicitement le code de terminaison de la tâche (cf. cours sur les signaux).

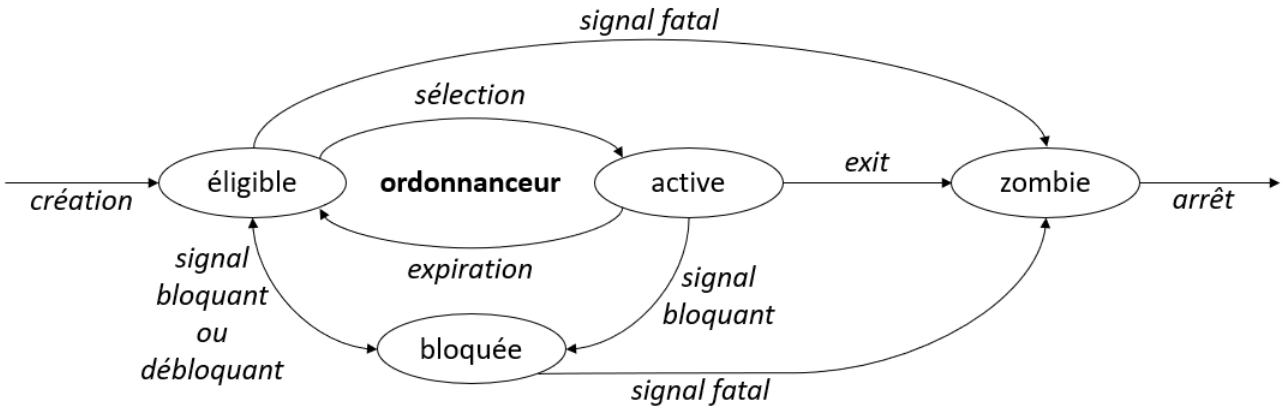

FIGURE 1 – États des processus.

Lorsqu'un processus attend un événement, il peut soit boucler sur un test non bloquant, soit réaliser un appel bloquant dont il ne ressortira que lorsque l'événement sera survenu. Ces deux stratégies, appelées respectivement attente active et attente passive, sont plus ou moins équivalentes du point de vue de la créativité du processus qui les met en oeuvre. Du point de vue du système, l'attente active correspond des transitions régulières entre l'état « tâche active » et celui de « tâche éligible » , et induit une consommation maximale en temps CPU, ce qui pénalise inutilement les autres tâches. Au contraire, l'attente passive place le processus dans l'état bloqué, ce qui évite toute occupation du CPU. La reprise de la tâche en attente peut même s'en trouver accélérée si elle dépend de l'avancement d'une autre tâche. Les développeurs d'applications doivent donc éviter au maximum les situations d'attente active.

Nota Bene : Il existe une tâche particulière, appelée « *idle* » , qui est chargée d'occuper voire de mettre en sommeil le CPU lorsqu'aucune tâche utile n'est éligible. Sous Linux, il s'agit d'une tâche noyau non montrée par ps. Sur un poste client standard, les tâches sont le plus souvent bloquées dans l'attente d'un événement, et seules des tâches de calcul intensif peuvent prétendre conserver longtemps le CPU. Il est don fréquent que la tâche *idle* occupe 95 % ou plus du temps CPU.

3.2. Gestion des processus en shell

La ommande **ps** permet d'afficher des informations sur tout ou partie de la liste des processus du système. Essayer par exemple **ps auxw**, **ps afx** ou encore **ps axl** (cf. **man ps** pour l'ensemble des possibilités offertes sur un OS particulier).

L'outil **pstree** présente la liste des processus sous forme d'arbre en fonction de leurs liens de parenté. **top** (cf. plus haut) permet également de visualiser de nombreuses informations. Le système de fichiers virtuel *proc*(*fs*), monté sur /proc , est une autre source d'informations. Celles relatives au processus n° <pid> sont accessibles dans /proc/<pid>/ (ou /proc/self/ pour le processus appelant). En particulier, sous Linux, on y trouve :

- **cmdline** : argv[] du processus ;
- **cwd** : pseudo-lien vers le répertoire de travail courant du processus ;
- **environ** : envp[] du processus ;
- **exe** : pseudo-lien vers le programme exécuté par le processus ;
- **fd/<n>** : pseudo-lien vers le fichier correspondant au descripteur numéro <n> dans la table des fichiers ouverts du processus (incidemment, ces liens désanonymisent les tubes anonymes...) ;
- **root** : pseudo-lien vers la racine relative du processus (chroot).

4. Propriétés des processus

Propriétés génériques

Un processus est composé de :

- **un contexte CPU** : l'état instantané du CPU au cours de l'exécution du programme ;
- **un contexte utilisateur** : la mémoire occupée par le programme (code, données, tas, bibliothèques, pile) ;
- **un contexte noyau** (appelé plus simplement « contexte » dans la suite), géré par le noyau de l'OS.

Le contexte CPU est sauvegardé dans le contexte noyau quand le processus n'est pas actif. Il est restauré par l'ordonnanceur au moment de redonner la main au processus. La gestion du programme en mémoire sera vue plus en détail dans les cours suivants. Le contexte regroupe de nombreuses informations indispensables au suivi de l'exécution du processus, notamment :

- son numéro d'identifiant (PID) ;
- un pointeur sur le contexte du père ;
- ses identités : uid et gid réels, effectifs, etc... (cf. plus bas) ;
- son état vis-à-vis de l'ordonnanceur (actif, éligible, bloqué ou zombie) ;
- les informations relatives sa priorité ;
- les informations relatives aux threads [2] : ontexte CPU (registres, pointeur d'ex ution, pointeur de pile, et .), propri t s sp i ques un thread ;
- les informations relatives aux signaux : masque, signaux en attente, fonctions de traitement, timers (cf. cours sur les signaux) ;
- les informations relatives la gestion des tâches par les terminaux : session, terminal de contrôle, groupe de processus (cf. ours sur les signaux) ;
- des informations relatives l'utilisation de VFS : le répertoire de travail courant, la racine relative (pour le chroot), le umask, etc... ;

2. Comme nous le verrons dans le cours sur les threads, certains systèmes d'exploitation gèrent les threads en couche applicative. Dans ce cas, le noyau n'a connaissance que d'un l d'ex ution par pro essus.

- la table des descripteurs de fichiers ouverts et de leurs attributs ;
- les verrous de fichiers BSD/SYSV/POSIX possédés, référencés indirectement à travers les descripteurs de fichiers ouverts ;
- la table des pages mémoire (cf. cours sur la mémoire) et de leurs propriétés ;
- des limitations sur l'utilisation des ressources système (cf. **getrlimit** et **setrlimit**) ;
- des statistiques sur l'exécution (cf. **times**, **getrusage**).

Les instructions POSIX suivantes permettent d'obtenir les PIDs d'un processus et de son père :

```
pid_t pid =  getpid(void)
pid_t ppid = getppid(void)
```

La première récupère le PID de l'appelant, la seconde le PID du père de l'appelant.

4.2. Gestion des identités

Sur la plupart des systèmes UNIX (POSIX, System V), chaque processus dispose de trois identités :

- son uid réel (uid), qui correspond normalement l'identité de l'utilisateur qui a lancé l'application et au bénéfice duquel elle tourne ;
- son uid effectif (euid), qui correspond aux privilèges effectifs du processus ;
- son uid sauvegardé (suid), qui correspond des privilèges effectifs qui ont été obtenus puis abandonnés provisoirement.

Nominalement, tous les uids sont identiques. Les changements interviennent lors du lancement d'un exécutable (notamment s'il y a un bit setuid), ou lors d'un appel une fonction de changement explicite d'identité. Les contraintes naturelles sur ces fonctions sont :

- si le processus est root : aucune ;
- sinon, il n'est possible d'effectuer que des permutations ou des copies entre uid, euid et suid. Les effets sont les suivants (si l'euid initial est non nul) :

Action	**uid**	**euid**	**suid**	**compatibilité**
exec bit s (u)	-	=u	=u	POSIX
exec sans bit s	-	-	=euid	POSIX
setuid(u1)	-	=u1	-	POSIX
seteuid(u1)	-	=u1	-	BSD
setreuid(u1,u2)	=u1	=u2	-	BSD
setresuid(u1,u2,u3)	=u1	=u2	=u3	spécifique

Nota bene : La descente de privilèges est une opération consistant pour un processus à perdre une partie de ses privilèges. Par exemple, un processus lancé en tant que *root* peut changer d'identité pour ne plus disposer des privilèges du super-utilisateur. La descente de privilège qu'un processus privilégié effectue avec **setuid** est irréversible. Elle contribue donc la tolérance aux intrusions : si un attaquant compromet le processus par la suite, les privilèges qu'il en tirera ne seront pas eux de *root*, ce qui réduit les conséquences d'une telle compromission sur le reste du système.
À l'inverse, le changement d'identité avec **seteuid** est réversible, et vise à prévenir une compromission : lorsqu'un processus réalise une action, il exploite par défaut l'ensemble des privilèges effectifs

qui lui sont attribués ; changer temporairement d'identité, par exemple le temps de répondre à la requête d'un utilisateur donné, lui évite d'utiliser accidentellement ses propres privilèges et de réaliser ainsi une action qui serait interdite à l'utilisateur (problème dit du « *confused deputy* »).

La gestion des groupes (*gid*, *egid*, *sgid* et éventuellement *fsgid*) s'effectue de la même façon. Noter que le privilige de s'auto-attribuer un groupe quelconque est réservé à l'euid 0 (utilisateur *root*) et non l'egid 0 (groupe *root*, qui n'est essentiellement pas privilégié au sens des appels système). Chaque processus dispose, en outre, d'une liste de groupes supplémentaires. Cette liste :

- est lisible par getgroups et modifiable (opération privilégiée) par **setgroups** ou sa surcouche **initgroups** (qui réinitialise la liste partir de la configuration courante des comptes, généralement dans */etc/group*) ;
- contient ou non l'egid selon les OS ;
- permet d'accéder aux fichiers (**open**, **exec**) via les permissions pour un de ses groupes ;
- permet de donner des fichiers (**chown**) à un de ses groupes (ce qui est transparent sur les OS qui autorisent systématiquement le don de fichiers) ;
- ne permet PAS directement (**setgid**) d'attribuer au processus un de ses groupes (interdiction faible : on peut créer un exécutable avec bit **setgid**, ou encore utiliser la ommande *shell* **newgrp** pour la contourner) ; cela présenterait peu d'intérêt dans la mesure où les groupes servent essentiellement à gérer les accès aux fichiers.

Une gestion astucieuse des groupes permet sur un système ne disposant pas de listes de contrôle d'accès (ACLs) de partager des fichiers entre utilisateurs travaillant sur un même projet ou encore de déléguer certains privilèges d'administration via des permissions sur des périphériques, des fichiers de configuration et/ou des exécutables avec bit setuid (en prenant en compte les risques d'escalade de privilèges [3] !).

4.3. Propriétés liées au VFS

Les principaux éléments liés à l'utilisation de VFS se manipulent par :

- **char *ret = getcwd(char *buf, size_t len)** : réupérer le nom du répertoire de travail courant (racine des chemins relatifs) (POSIX) (équivalent *shell* : **pwd**) ;
- **int ret = chdir(const char *nom)** : changer le répertoire de travail courant (POSIX) (équivalent *shell* : **cd**) ;
- **int ret = fchdir(int fd)** : changer le répertoire de travail courant en utilisant un descripteur de répertoire ouvert (BSD + System V) ;
- **int ret = chroot(const char *nom)** : changer la racine (/) relative du processus, qui ne pourra plus désigner que des fichiers situés, dans l'arborescence VFS, sous cette racine relative [4] (réservé à *root*) (BSD + System V) (équivalent shell : **chroot**) ;
- **mode_t old_mask = umask(mode_t new_mask)** : changer le *umask* courant, c'est à dire les permissions retirer lors de la création de nouveaux fichiers (cf. **open**, **creat**, **mkdir**, et .) (POSIX) (équivalent *shell* : **umask**) ; **chmod** n'est quant lui pas limité par le **umask** ;

3. Ainsi, l'écriture sur les fichiers de configuration résseau permet généralement de programmer l'exécution de commandes arbitraires sous le compte *root* ; l'écriture bas niveau sur le disque dur permet de changer les permissions ou le contenu de ses fichiers ; le détournement d'un programme avec bit *setuid* peut permettre d'obtenir un *shell root* ; etc...00

4. La racine relative remplace la racine de VFS dans l'interprétation des chemins absolus. Pour le processus concernés, elle a la propriété d'une racine (.. = .).

chroot constitue une fonction de sécurité intéressante, dans la mesure où elle permet d'enfermer un processus dans une « cage » minimaliste au sens du système de fichiers. Dans la pratique, il faut construire soigneusement la cage, ajuster le répertoire de travail courant et restreindre les privilèges du processus (ne pas rester *root*, mais prendre une identité dédiée) pour limiter les risques qu'un attaquant ayant pris le contrôle du processus ne parvienne sortir de la cage.

5. Mise en oeuvre pratique des processus

5.1. Création d'un nouveau processus

La création d'un nouveau processus se produit :

- en temps normal, lorsqu'un processus utilisateur se duplique via un appel à la fonction fork [5] ;
- exceptionnellement, l'initiative du noyau (pour lancer le premier processus utilisateur, **init**, de pid 1, ou par exemple pour gérer le branchement chaud d'un périphérique sous Linux).

Dans le cas de **fork**, le nouveau processus (fils) est au départ une copie conforme du père, tant du point de vue du contenu de la mémoire qu'il occupe que de celui du contexte géré par le noyau (voir ci-dessous pour les exceptions). Les deux processus continuent l'exécution du code courant (instruction suivant le **fork**), mais de façon indépendante.
En réalité, lors d'un **fork**, le système s'arrange pour n'effectuer les copies de pages mémoire que lorsqu'elles sont nécessaires, plutôt que de façon systématique lors du **fork**. La technique couramment employée est elle du *copy on write* : la page n'est copiée qu'à la première tentative d'accès en écriture, ce qui permet d'alléger le **fork** et d'économiser de la RAM. Ce mode de fonctionnement permet en particulier d'optimiser le cas courant où **fork** est suivi d'un *exec*. Ce dernier réinitialisera complètement l'espace mémoire, et au une copie inutile n'aura eu lieu.
L'appel à la fonction s'effectue ainsi :

- **pid_t new_pid = fork(void)** : création d'un nouveau processus (POSIX).

En cas d'échec, le « père » lit -1 et il n'y a pas de nouveau processus. Sinon, le père lit le pid du fils (un entier strictement positif) et le fils lit 0, ce qui permet à chacun de déterminer immédiatement qui il est et d'adapter son comportement au rôle qui lui est attribué.

Quelques points noter :

- les variables (données) font partie des projections mémoires, et sont copiées. Elles ne sont donc pas liées (la modification d'une variable globale du père n'affecte pas la copie vue par le fils, et vice-versa) ;
- il existe un lien particulier entre le père et le fils (cf. **wait**, plus bas, et le cours sur les signaux) ;
- le pid du fils diffère de celui du père (même chose pour le ppid) ;
- la table des descripteurs de fichiers ouverts du fils est une copie de celle du père ; ces copies sont liées comme si elles avaient été générées par dup (cf. cours précédents : il y a partage des structures système décrivant les propriétés des accès aux fichiers), donc les déplacements dans les fichiers ouverts avant **fork** sont communs [6] ;
- la plupart des propriétés du père sont héritées par le fils (priorité, privilèges, contexte VFS, limites, traitement des signaux et des terminaux), les principales exceptions étant remises zéro pour le fils :

5. Il existe aussi des variantes comme **vfork**, **clone** (Linux).
6. Dans la pratique, **argv** et **envp** sont des **const char *const tab[]**. Le dernier pointeur de chaque tableau doit être à NULL

les verrous de fichiers SYSV/POSIX (les verrous BSD sont quant à eux partagés, plutôt que dupliqués, cf. cours précédent),

les threads (cf. cours suivant),

les protections mémoire de type **mlock** (cf. cours sur la mémoire),

les signaux en attente, les timers (cf. cours sur les signaux),

les statistiques d'exécution.

L'espace mémoire n'étant pas partagé, les deux processus doivent utiliser des moyens externes de communication s'ils souhaitent s'échanger des informations :

- comme des processus qui ne se connaissent pas : utilisation des fichiers du FS, envoi de signaux, communication par *sockets*, etc... ;
- en utilisant l'héritage commun : partage d'un tube anonyme, d'une paire de *sockets* locales anonymes, d'un segment de mémoire partagée, etc... ;
- en utilisant le code de retour (terminaison) du fils, que le père peut récupérer.

5.2 . Exécution d'un nouveau programme

La fonction **fork** ne permet que de dupliquer un processus existant. L'exécution d'un code différent nécessite l'appel à une fonction de type **exec**, qui charge un nouveau programme et l'exécute en écrasant le programme courant tout en conservant certains élèments du contexte. La fonction POSIX de base est :

- int ret = execve(const char *nom_prog, char *const argv[], char *const envp[])

Cette fonction va permettre le remplacement du programme courant par nom_prog (chemin complet ; permission exécution requise), son **main** sera exécuté avec les arguments argv et les variables d'environnement envp (le prototype complet du **main** est, sur la plupart des systèmes :

- int main(int argc, char *argv[], char *envp[])

Sauf échec lors de l'appel, le code qui suit **execve** n'est jamais exécuté puisque le programme courant est remplacé par le nouveau programme. Lorsque le processus se terminera, il renverra le code de terminaison de ce nouveau programme.

Lors de l'appel à une fonction **exec** (voir le cours sur les exécutables pour plus de détails) :

1. l'OS vérifie les permissions « x » et réagit en fonction du type d'exécutable (script, format binaire reconnu), déterminé d'après les premiers octets du fichier ;
2. les éventuels bits « s » changent les identités effectives du processus appelant (pour un script, ils sont ignorés par la plupart des OS), qui sont ensuite sauvegardées ;
3. l'espace mémoire (code, données, tas, bibliothèques, pile) et les threads (cf. cours suivant) sont complètement réinitialisés ;
4. dans le contexte noyau, sont notamment conservés :

   le pid et le ppid,

   la priorité,

   les identités réelles du processus et ses groupes supplémentaires (le reste dépend des bits « s »),

   les paramètres liés à VFS (répertoire de travail, racine relative, *umask*, etc...),

les éléments relatifs aux signaux (timers, listes des signaux masqués, des signaux en attente et des signaux ignorés, sauf éventuellement SIGCHLD, f. cours sur les signaux), l'exception des fonctions de traitement enregistrées,

les éléments relatifs aux terminaux (session, terminal de contrôle, groupe de processus),

les limites sur l'utilisation des ressources,

les statistiques d'exécution.

5. dans la plupart des cas, la table des fichiers ouverts et les verrous de fichiers sont aussi conservés (cf. plus bas).

5.3. Terminaison d'un processus

Le mécanisme normal de terminaison d'un processus est le suivant :

1. le processus réalise un appel **exit** pour demander au noyau de le terminer en précisant un code de retour ; seul l'octet de poids faible est significatif ;
2. le père récupère le code de retour via un appel à une fonction **wait**.

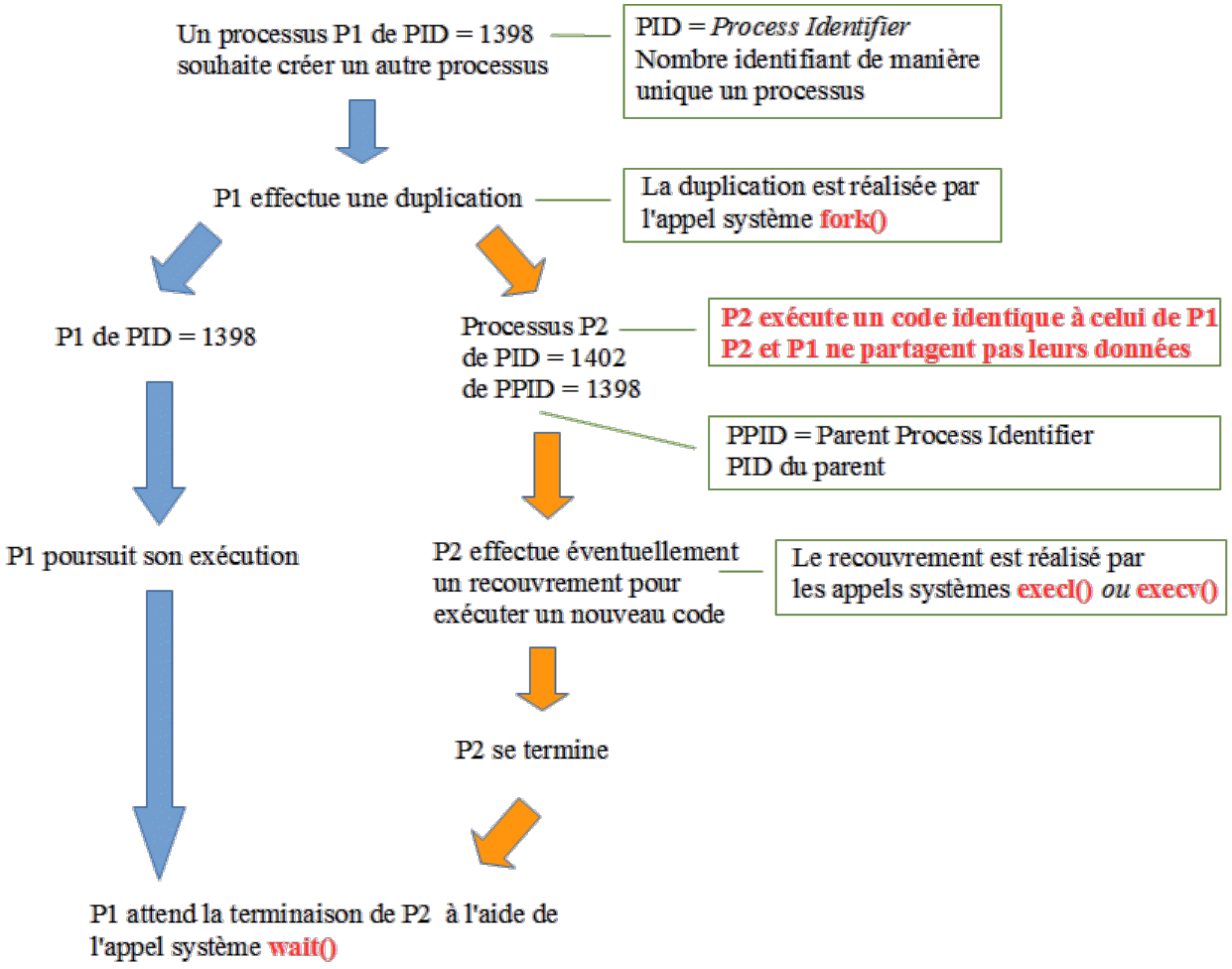

FIGURE 2 – Création d'un processus.

Rappel : la hiérarchie des processus est défini par les règles suivantes :

- le processus initial est *init* (pid 1). Il est créé à l'initialisation par le système d'exploitation ;
- tout processus a un processus parent sauf le processus initial *init* ;
- pour arrêter une machine, *init* doit être terminé. En faisant cela, il met fin automatiquement à tous les processus fils.

5.4. Application : réalisation du pipe *shell*

Le pseudo-code qui suit montre le principe de fonctionnement du pipe *shell* (exécution d'une ligne de type « md1 | md2 » via deux sous-shells). Pour l'implémenter réellement, il faut utiliser les prototypes exacts des fonctions et traiter proprement les cas d'erreur.

```
[interpréteur]
 pipe(&fd_r, &fd_w)
 fork() ------------------------------------
   |                                        |
[père]                                   [fils1]
 close(fd_w)                              close(fd_r)
 fork()--------------------               dup2(fd_w, 1)
   |                    |                 close(fd_w)
[père]                [fils2]              exec("cmd1")
 close(fd_r)           dup2(fd_r,0)
 wait(&status)         close(fd_r)
 wait(&status)         exec("cmd2")
```

FIGURE 3 – Pseudo-code de la commande *shell* **pipe**.

6. Fonctions liées à la gestion de l'ordonnanceur

6.1. Interactions avec l'ordonnanceur UNIX depuis le *shell*

La commande **top** permet d'observer dynamiquement les processus en cours d'exécution et de les classer par taux d'occupation des ressources (CPU, RAM, etc...). Sont également affichés les priorités et les temps d'exécution des différents processus, ainsi que des statistiques globales sur l'occupation de la mémoire et sur le *swap* (cf. cours sur la mémoire). De nombreuses indications supplémentaires peuvent être affichées ou utilisées comme critère de tri (nom des appels systèmes bloquants en attente, etc...), cf. **man top**.

La commande **nice** permet de modifier la priorité (gentillesse) d'un programme à lancer. La valeur par défaut est 0, les valeurs plus prioritaires sont négatives et réservées à *root*, alors que les valeurs positives sont moins prioritaires et accessibles tous (cf. **man nice** pour les valeurs admises pour chaque OS) :

```
nice -n <priorite> <commande>
```

renice <**priorite**> **[-p]** <**pid**> permet quant à elle d'ajuster la priorité d'un processus déjà lancé. Seul **root** peut augmenter la priorité d'un processus (c'est-à-dire diminuer la valeur de *nice*).

Ce document est inspiré du « Cours Système n°4 : les processus » délivré par monsieur Olivier Levillain dans les cours dispensés à l'Agence nationale de la sécurité des systèmes d'information (Novembre 2012).