Fondements des systèmes d’exploitation

Sofiane Khalfallah

ISSAT Sousse

1ING-GL

Initiation à Unix

- Unix System Overview
  - Symbolinc Links
- Standard I/O Library
- Process
- Signals
- Threads
- Advanced I/O
- Interprocess Communication
- Network IPC: Sockets

Introduction

Développement en C sous Uinux

Cinq types d'utilitaires :

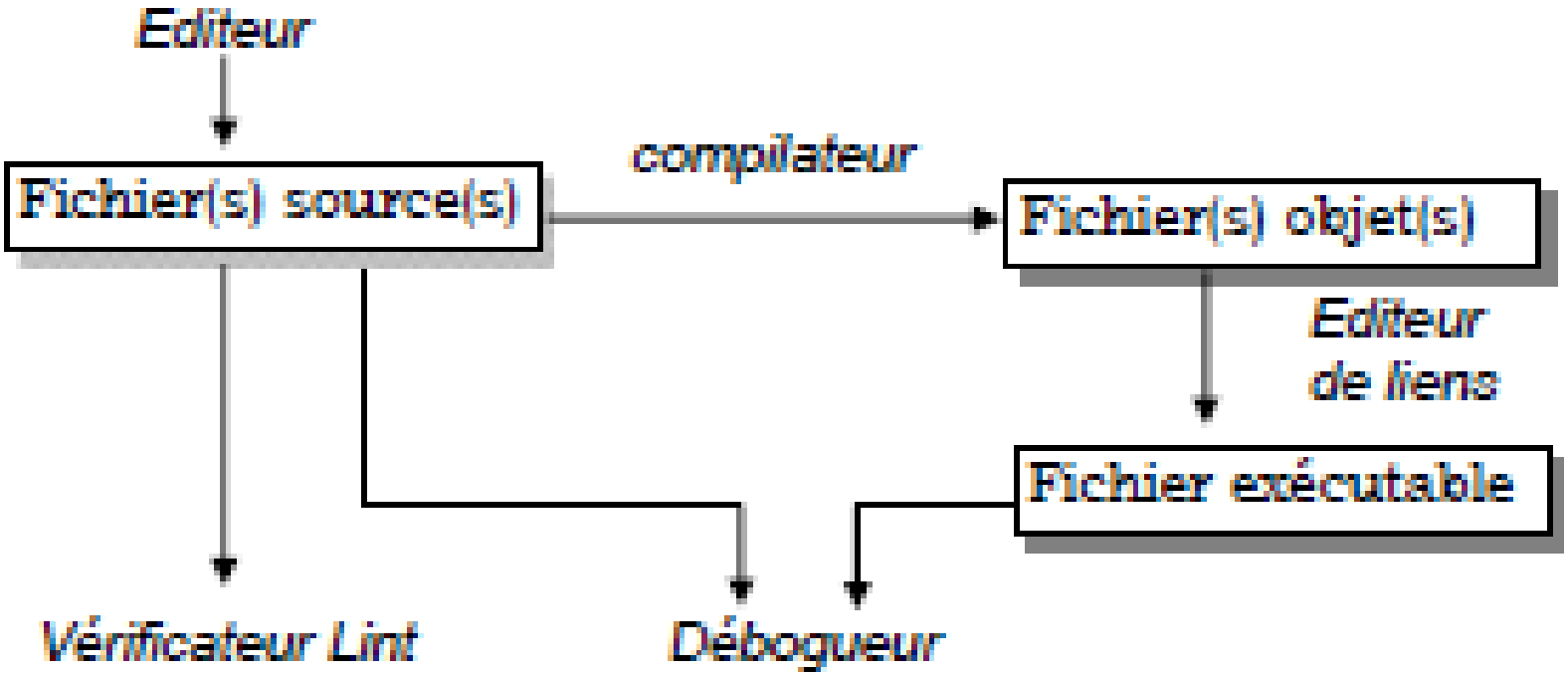
- L'éditeur de texte, qui est à l'origine de tout le processus de développement applicatif. Il nous permet de créer et de modifier les fichiers source.

- Le compilateur, qui permet de passer d'un fichier source à un fichier objet. Cette transformation se fait en réalité en plusieurs étapes grâce à différents composants

- L'éditer de liens, qui assure le regroupement des fichiers objet provenant des différents modules et les associe avec les bibliothèques utilisées pour l'application. Nous obtenons ici un fichier exécutable.
- Le débogueur, qui peut alors permettre l'exécution pas à pas du code, l'examen des variables internes, etc. Pour cela, il a besoin du fichier exécutable et du code source.

- Notons également l'emploi éventuel d'utilitaires annexes travaillant à partir du code source, comme le vérificateur Lint, les enjoliveurs de code, les outils de documentation automatique, etc.

Processus de développement en C



Editeurs de texte : Vi et Emacs

- *Emacs est théoriquement un éditeur de texte, mais des possibilités d'extension par l*'intermédiaire de scripts Lisp en ont fait une énorme machine capable d'offrir l'essentiel des commandes dont un développeur peut rêver.

- *Vi est beaucoup plus léger, il offre nettement moins de fonctionnalités et de possibilités* d'extensions que *Emacs. Les avantages de Vi sont sa disponibilité sur toutes les plates-formes* Unix et la possibilité de l'utiliser même sur un système très réduit pour réparer des fichiers de configuration.

- *Vi et Emacs peuvent fonctionner sur un terminal texte, mais ils sont largement plus* simples à utiliser dans leur version fenêtrée X11.

Compilateur gcc

- L'invocation de gcc se fait avec les arguments suivants :
  - Les noms des fichiers C à compiler ou les noms des fichiers objet à lier. On peut en effet procéder en plusieurs étapes pour compiler les différents modules d'un projet, retardant l'édition des liens jusqu'au moment où tous les fichiers objet seront disponibles.
  - Éventuellement des définitions de macros pour le préprocesseur, précédées de l'option

  —D. Par exemple —D_XOPEN_SOURCE=500 est équivalent à une directive #Define

  _XOPEN_SOURCE 500 avant l'inclusion de tout fichier d'en-tête.

Compilateur gcc

- Éventuellement le chemin de recherche des fichiers d'en-tête (en plus de /usr/include), précédé de l'option –I avec par exemple :

–I/usr/X11R6/include.

- Éventuellement le chemin de recherche des bibliothèques supplémentaires (en plus de /lib et /usr/lib), précédé de l'option –L avec par exemple –L/usr/X11R6/1ib/.
- Le nom d'une bibliothèque supplémentaire à utiliser lors de l'édition des liens, précédé du préfixe -l. Il s'agit bien du nom de la bibliothèque, et pas du fichier.
- On peut préciser le nom du fichier exécutable, précédé de l'option –o.

Compilateur gcc

- Enfin, plusieurs options simples peuvent être utilisées, dont les plus courantes sont :

Option	Argument	But
-E		Arrêter la compilation après le passage du préprocesseur, avant le compilateur.
-S		Arrêter la compilation après le passage du compilateur, avant rassembleur.
-c		Arrêter la compilation après l'assemblage, laissant les fichiers objet disponibles.
-W	Avertissement	Valider les avertissements (*warnings*) décrits en arguments. Il en existe une multitude. mais l'option la plus couramment utilisée est -Wall , pour activer tous les avertissements.
-pedantic		Le compilateur fournit des avertissements encore plus rigoureux qu'avec -Wall, principalement orientés sur la portabilité du code.
-g		Inclure dans le code exécutable les informations nécessaires pour utiliser le débogueur. Cette option est généralement conservée jusqu'au basculement du produit en code de distribution.
-O	0 à 3	Optimiser le code engendré. Le niveau d'optimisation est indiqué en argument (0= aucune). Il est déconseillé d'utiliser simultanément l'optimisation et le débogage.

Exemple : gcc

- Les combinaisons d'options les plus couramment utilisées sont donc
  - gcc -Wall -pedantic -g fichier1.c -c
  - gcc -Wall -pedantic -g fichier2.c -c
- Qui permettent d'obtenir deux fichiers exécutables qu'on regroupe ensuite ainsi :
  - gcc fichier1.o fichier2.o -o resultat
- On peut aussi effectuer directement la compilation et l'édition des liens :
  - gcc -Wall -pedantic -g fichier1.c fichier2.c -o resultat
- Lorsque le code a atteint la maturité nécessaire pour basculer en version de distribution
  - gcc -Wall -DNDEBUG -02 fichier1.c fichier2.c -o resultat

Utilitaires divers

- L'outil **grep** est essentiel pour un programmeur, car il permet de rechercher une chaîne de caractères dans un ensemble de fichiers. Sa page de manuel documente ces nombreuses fonctionnalités, et l'emploi des

  expressions régulières pour préciser le motif à rechercher.
- La commande **find** recherche tous les fichiers réguliers (-type f) de manière récursive à partir du répertoire en cours (.), et envoie les résultats à xargs. Cet utilitaire les regroupe en une liste d'arguments qu'il transmet à grep pour y rechercher la chaîne demandée.

Utilitaires divers

- $ cd /usr/src/linux
- $ find . -type f | xargs grep ICMPV6_ECHO_REQUEST

./net/ipv6/icmp.c: else if (type >=
ICMPV6_ECHO_REQUEST &&

./net/ipv6/icmp.c:
(&icmpv6_statistics.Icmp6InEchos)[type-
ICMPV6_ECHO_REQUEST]++;

./net/ipv6/icmp.c: case ICMPV6_ECHO_REQUEST:

./include/linux/icmpv6.h:#define
ICMPV6_ECHO_REOUEST 128

$

Utilitaires divers

- L'utilitaire **diff** permet de comparer intelligemment deux fichiers et indique les portions modifiées entre les deux.

```
$ diff hello.3.c hello.4.c
3c3,4
< int main(int argc, char *argv[])
---
> int
> main(int argc, char *argv[])
```

Processus Unix

Processus Unix

- Sous Unix, toute tâche qui est en cours d'exécution est représentée par un **processus. Un** processus est une entité comportant à la fois des données et du code.
- On peut considérer un processus comme une unité élémentaire en ce qui concerne l'activité sur le système.

Commande Ps

- On peut examiner la liste des processus présents sur le système à l'aide de la commande **ps**, et plus particulièrement avec ses options **ax**, qui nous permettent de voir les processus endormis, et ceux qui appartiennent aux autres utilisateurs.

Exemple commande Ps

```
$ ps ax
PID  TTY  STAT  TIME  COMMAND
    1  ?  S    0:03 Init
    2  ?  SW   0:03 (kflushd)
    3  ?  SW<  0:00 (kswapd)
    4  ?  SW   0:00 (nfsiod)
    5  ?  SW   0:00 (nfsiod)
    6  ?  SW   0:00 (nfsiod)
    7  ?  SW   0:00 (nfsiod)
   28  ?  S    0:00 /sbin/kerneld
  194  ?  S    1:26 syslogd
  203  ?  S    0:00 klogd
  225  ?  S    0:00 crond
  236  ?  SW   0:00 (inetd)
  247  ?  SW   0:00 (lpd)
  266  ?  S    0:00 (sendmail)
  278  ?  S    0:00 gpm -t PS/2
  291  1  SW   0:00 (mingetty)
    [...]
  626  ?  SW   0:00 (axnet)
25896  ?  SW   0:00 (.xsession)
25913  ?  S    0:15 xfwm
25915  ?  S    0:04 /usr/X11R6/bin/xfce 8 4 /var/XFCE/system.xfwmrc 0 8
25918  ?  S    0:03 /usr/X11R6/bin/xfsound 10 4 /var/XFCE/system.xfwmrc
25919  ?  S    0:00 /usr/X11R6/bin/xfpager 12 4 /var/XFCE/system.xfwmrc
29434  ?  S    1:56 /usr/local/applix/axdata/axmain -helper 29430 6 10 -
29436  ?  SW   0:00 (applix)
29802  p0 S    0:00 -bash
29970  ?  S    0:00 xplaycd
29978  ?  S    0:00 xmixer
30550  p1 S    0:00 -bash
31144  p1 R    0:00 ps ax
$
```

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( ),
- fork() va dupliquer le processus appelant. Au retour de cet appel-système, deux processus iden tiques continueront d'exécuter le code à la suite de fork( ).

Identification par le PID

- La différence essentielle entre ces deux processus est un numéro d'identification. On distingue ainsi le processus original, qu'on nomme traditionnellement le processus père, et la nouvelle copie le processus fils.
- L'appel-système fork( ) est déclaré dans <unistd.h>, ainsi :

$pid_t fork(void);

getpid( )

- Pour connaître son propre identifiant PID, on utilise l'appel-système getpid( ), qui ne prend pas d'argument et renvoie une valeur de type pid_t. Il s'agit. bien entendu, du PID du processus appelant :

$ pid_t getpid (void);

- Getpid() renvoie une valeur de type pid_t, qui vaut zéro si on se trouve dans le processus fils, est négative en cas d'erreur, et correspond au PID du fils si on se trouve dans le processus père.

getppid( )

- Le processus fils peut aisément accéder au PID de son père (noté PPID pour *Parent PID) grâce à l'appel-système* getppid( ), déclaré dans <unistd.h> :

$ pid_t getppid (void);

- Cette routine se comporte comme getpid( ) , mais renvoie le PID du père du processus appelant. Par contre, le processus père ne peut connaître le numéro du nouveau processus créé qu'au moment du retour du fork( ).

Hiérarchie des processus

- On peut examiner la hiérarchie des processus en cours sur le système avec le champ PPID de la commande ps axj:

```
$ ps axj
PPID   PID  PGID  SID  TTY  TPGID  STAT  UID  TIME COMMAND
   0     1     0    0  ?     -1  S       0  0:03 init
   1     2     1    1  ?     -1  SW      0  0:03 (kflushd)
   1     3     1    1  ?     -1  SW<     0  0:00 (kswapd)
   1     4     1    1  ?     -1  SW      0  0:00 (nfsiod)
   [...]
   1   296   296  296  6    296  SW      0  0:00 (mingetty)
 297   301   301  297  ?     -1  S       0 45:56 usr/X11R6/bin/X
 297 25884 25884  297  ?     -1  S       0  0:00 (xdm)
```

Exemple : création d'un processus fils

- Lors de son exécution, ce programme fournit les informations suivantes :

```
$ ./exemple fork
Père : PID=31382, PPID=30550, PID fils=31383
Fils : PID=31383, PPID=31382
$
```

- Le PPID du processus père correspond au shell.

Signaux Unix

Principe

- Un processus peut envoyer sous certaines conditions un signal à un autre processus (ou à lui-même). Un signal peut être imaginé comme une sorte d'impulsion qui oblige le processus cible à prendre immédiatement (aux délais dus à l'ordonnance-ment près) une mesure spécifique.
- Le destinataire peut soit ignorer le signal. Soit le capturer c'est-à-dire dérouter provisoirement son exécution vers une routine particulière qu'on nomme gestionnaire de signal. Soit laisser le système traiter le signal avec un comportement par défaut.

Norme Posix

- Certains signaux font partie d'une classe nouvelle, les signaux temps-réel, définis par la norme Posix

```
#include <signal.h>

#ifndef NSIG
    #ifndef _NSIG
        #error "NSIG et _NSIG indéfinis"
    #else
        #define NSIG _NSIG
#endif #endi
```

Signal SIGIO

- SIGIO est envoyé lorsqu'un descripteur de fichier change d'état et permet la lecture ou l'écriture. Il s'agit généralement de descripteurs associés à des tubes, des sockets de connexion réseau, ou un terminal.
- L'utilisation de SIGIO permet à un programme d'accéder à un traitement d'entrée sortie totalement asynchrone par rapport au déroulement du programme principal.
- Les données arrivant pourront servir par exemple à la mise à jour de variables globales consultées régulièrement dans le cours du programme normal.

Signal SIGINT

- Ce signal est émis vers tous les processus du groupe en avant-plan lors de la frappe d'une touche particulière du terminal : la touche d'interruption.
- Il s'agit habituellement de Contrôle-C.
- L'affectation de la commande d'interruption à la touche choisie peut être modifiée par l'intermédiaire de la commande stty

Signal SIGQUIT

- Comme SIGINT, ce signal est émis par le pilote de terminal lors de la frappe d'une touche particulière : la touche QUIT.
- Celle-ci est affectée généralement à Contrôle-\ (ce qui nécessite sur les claviers français la séquence triple Ctrl-AltGr-\).
- SIGQUIT termine par défaut les processus qui ne l'ignorent pas pas et ne le capturent pas, mais en engendrant en plus un fichier d'image mémoire (core).

Signal SIGSTOP

- SIGSTOP est le deuxième signal ne pouvant être ni capturé ni ignoré, comme SIGKILL. Il a toutefois un effet nettement moins dramatique que ce dernier, puisqu'il ne s'agit que d'arrêter temporairement le processus visé. Celui passe à l'état stoppé.

Signal SIGKILL

- SIGKILL est l'un des deux seuls signaux (avec SIGSTOP) qui ne puisse ni être capturé ni être ignoré par un processus.
- A sa réception, tout processus est immédiatement arrêté. C'est une garantie pour s'assurer qu'on pourra toujours reprendre la main sur un programme particulièrenient rétif. Le seul processus qui ne puisse pas recevoir SIGKILL est init, le processus de PID 1.
- SIGKILL est traditionnellement associé à la valeur 9, d'où la célèbre ligne de commande «kill -9 xxx» , équivalente à « Kill -KILL xxxx ». On notera que l'utilisation de SIGKILL doit être considérée comme un dernier recours. le processus ne pouvant se terminer propre-ment.
- On préfèrera essayer par exemple SIGQUIT ou SIGTERM auparavant.

Émission d'un signal sous Linux

- Pour envoyer un signal à un processus, on utilise l'appel-système kill( ), Cet appel système est déclaré ainsi :

$ int kill (pid_t pid, int numero_fils);

- Le premier argument correspond au PID du processus visé, et le second au numéro du signal à envoyer. Rappelons qu'il est essentiel d'utiliser la constante symbolique correspondant au nom du signal et non la valeur numérique directe.

Réception des signaux avec l'appel-système signal( )

- Un processus peut demander au noyau d'installer un gestionnaire pour un signal particulier, c'est-à-dire une routine spécifique qui sera invoquée lors de l'arrivée de ce signal.
- Le processus peut aussi vouloir que le signal soit ignoré lorsqu'il arrive, ou laisser le noyau appliquer le comportement par défaut (souvent une terminaison du programme).

Signal(), sigaction( )

- Pour indiquer son choix au noyau, il y a deux possibilités :
  - L'appel-système signal( ), défini par Ansi C et Posix.1, présente l'avantage d'être très simple (on installe un gestionnaire en une seule ligne de code), mais il peut parfois poserdes problèmes de fiabilité de délivrance des signaux et de compatibilité entre les divers systèmes Unix.
- L'appel-système sigaction( ) est légèrement plus complexe puisqu'il implique le remplissage d'une structure, mais il permet de définir précisément le comportement désiré pour le gestionnaire, sans ambiguïté suivant les systèmes puisqu'il est complètement défini par Posix.

Signal()

- L'appel-système signal( ) présente un prototype suivant :

$ void (*signal (int numero_signal, void (*gestionnaire) (int))) (int);

- On peut décomposer le prototype, en créant un type intermédiaire correspondant à un pointeur sur une routine de gestion de signaux :

$ typedef void (*gestion_t)(int);

- Le prototype de signal( ) devient :

$ gestion_t signal (int numero_signal, gestion_t gestionnaire);

Signal()

- En d'autres termes, signal( ) prend en premier argument un numéro de signal. Bien entendu, il faut utiliser la constante symbolique correspondant au nom du signal, jamais la valeur numérique directe.
- Le second argument est un pointeur sur la routine qu'on désire installer comme gestionnaire de signal.
- L'appel-système nous renvoie un pointeur sur l'ancien gestionnaire, ce qui permet de le sauvegarder pour éventuellement le réinstaller plus tard.

Signal()

- Il existe deux constantes symboliques qui peuvent remplacer le pointeur sur un gestionnaire, SIG_IGN et SIG_DFL, qui sont définies dans <signal.h>.
- La constante SIGIGN demande au noyau d'ignorer le signal indiqué. Par exemple l'appel système signal(SIGCHLD, SIG_IGN) — déconseillé par Posix — a ainsi pour effet sous Uinux d'éliminer directement les processus fils qui se terminent, sans les laisser à l'état zombie.
- Avec la constante SIG_DFL, on demande au noyau de réinstaller le comportement par défaut pour le signal considéré.
- Si l'appel-système signal( ) échoue, il renvoie une valeur particulière, elle aussi définie dans <signal.h> : SIG_ERR.

Exemple signal()

- Nous allons pouvoir installer notre premier gestionnaire de signal. Nous allons tenter de capturer tous les signaux. Bien entendu, signal( ) échouera pour SIGKILL et SIGSTOP.

- Pour tous les autres signaux, notre programme affichera le PID du processus en cours, suivi du numéro de signal et de son nom. Il faudra disposer d'une seconde console (ou d'un autre Xterm) pour pouvoir tuer le processus à la fin.

Ordonnancement des processus

Plan (1)

- Nous étudierons tout d'abord les différents états dans lesquels un processus peut se trouver, ainsi que l'influence du noyau sur leurs transitions.
- Nous analyserons ensuite les méthodes simples permettant de modifier la priorité d'un processus par rapport aux autres.

Plan (Suite)

- Enfin, nous observerons les fonctionnalités définies par la norme Posix.1b, qui permettent de modifier l'ordonnancement des processus, principalement dans l'esprit d'un fonctionnement temps-réel.

Etats d’un processus

- un processus peut se trouver dans un certain nombre d'états différents, en fonction de ses activités.
- Ces états peuvent être examinés à l'aide de la commande ps ou en regardant le contenu du pseudo-fichier /proc/<pid>/status.
- Ce dernier contient en effet une ligne «State:...» indiquant l'état du processus.

États d’un processus

État	Anglais	Signification
Exécution	Running (R)	Le processus est en cours de fonctionnement, il effectue un travail actif.
Sommeil	Sleeping (S)	Le processus est en attente d'un événement extérieur. Il se met en sommeil.
Arrêt	Stopped (T)	Le processus a été temporairement arrêté par un signal. Il ne s'exécute plus et ne réagira qu'à un signal de redémarrage.
Zombie	Zombie (Z)	Le processus s'est terminé, mais son père n'a pas encore lu son code de retour.

État exécution

- Le processus s'exécute normalement, il a accès au processeur de la machine et avance dans son code exécutable.

État sommeil

- Notons enfin qu'il existe deux types de sommeil : interruptible et ininterruptible.
- Dans le premier état, un processus peut être réveillé par l'arrivée d'un signal.
- Dans le second cas, le processus ne peut être réveillé que par une interruption matérielle reçue par le noyau.

État stoppé (1)

- Lorsqu'un processus reçoit un signal SIGSTOP, il est arrêté temporairement mais pas terminé définitivement. Ce signal peut être engendré par l'utilisateur (avec /bin/kill par exemple), par le terminal (généralement avec la touche Contrôle-Z), ou encore par un débogueur comme gdb qui interrompt le processus pour l'exécuter pas à pas.

État stoppé (2)

- Le signal SIGCONT permet au processus de redémarrer ; il est déclenché soit par /bin/kill, soit par le shell (commandes internes bg ou fg), ou encore par le débogueur pour reprendre l'exécution.

État Zombie (1)

- Enfin, un processus qui se termine doit renvoyer une valeur à son père. Cette valeur est celle qui est fournie à l'instruction return( ) de la fonction main( ) ou dans l'argument de la fonction exit( ).
- Tant que le processus père n'a pas lu cette valeur, le fils reste dans un état intermédiaire entre la vie et la mort, toutes ses ressources ont été libérées, mais il conserve une place dans la table des tâches sur le système.

État Zombie (2)

- On dit qu'il est à l'état Zombie.

État Zombie (3)

- Si le processus père ignore explicitement le signal SIGCHLD, le processus meurt tout de suite, sans rester à l'état zombie.
- Si, au contraire, le processus père ne lit jamais la valeur de retour et laisse à SIGCHLD son comportement par défaut, le fils restera à l'état zombie indéfiniment.

État Zombie (4)

- Si le processus père ignore explicitement le signal SIGCHLD, le processus meurt tout de suite, sans rester à l'état zombie.
- Si, au contraire, le processus père ne lit jamais la valeur de retour et laisse à SIGCHLD son comportement par défaut, le fils restera à l'état zombie indéfiniment.

État Zombie (5)

- Lorsque le processus père se termine à son tour, le fils orphelin est adopté par le processus numéro 1, init, qui lit immédiatement sa valeur de retour (même s'il n'en a aucune utilité), permettant enfin à ce processus de mourir enfin.

États successifs d'un processus