Langage de commande bash sous Linux

Basile Starynkévitch
basile@starynkevitch.net et basile.starynkevitch@cea.fr

intervenant à l'ENSTA

septembre 2021

Les opinions sont personnelles

git 8aeb77e9faaddf5d

Plan

Introduction

À connaître et méditer par ailleurs

- audience: **vous avez déjà programmé** (par exemple en classes préparatoires).
- aspects légaux de l'informatique (voir articles 323 et suivants du Code Pénal), éthiques (fichage des personnes, reconnaissance faciale, armes létales robotiques autonomes, vidéosurveillance automatisée, droit à l'oubli, décisions punitives automatisées et leur explicabilité ...) et droit à la déconnexion. **Réglement Général sur la Protection des Données** www.cnil.fr/fr/comprendre-le-rgpd
- aspects économiques: crise d'approvisionnement en semiconducteurs en 2021. Dépendance aux pays et continents extérieurs.
- aspects écologiques: pollution et consommation énergetique du traitement informatique, gestion des déchets électroniques, optimisation des consommations.
- aspects psychologiques et de santé: dépendance excessive aux écrans d'ordinateur, posture physique, fatigue visuelle.

Introduction

importance des bogues et efficacité

- **ominiprésence des bogues** et limitations théoriques (machine de Turing, indécidabilité)
- **explosion combinatoire** : problème du voyageur de commerce
- certains bogues ont tué
- *approche compositionnelle et modulaire* du développement: un logiciel est un assemblage de composants logiciels plus petits, dont certains sont réutilisables. `bash` sert souvent à cet assemblage.
- **importance de la documentation** et des *conventions* de codage (à documenter).

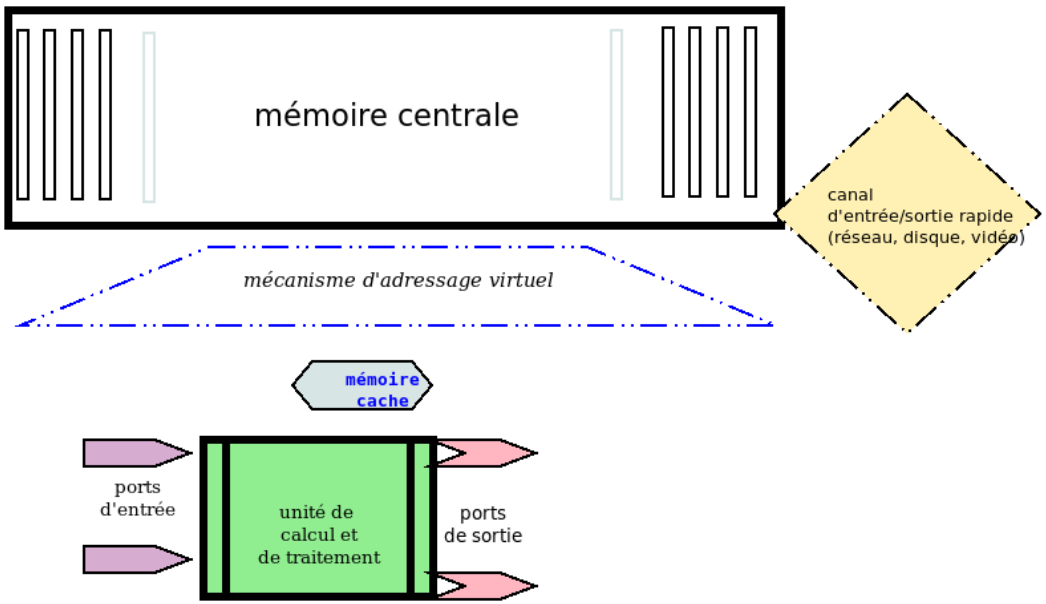
Conseil

sauvegardez votre travail (sur clef USB, le réseau, ...). Utilisez à bon escient un logiciel de gestion de versions comme `git`.

Introduction

Notion d'architecture matérielle des ordinateurs

Architectures de Von Neumann : les données et les instructions machine sont dans une mémoire commune. Donc le processeur peut modifier les instructions qu'il exécutera plus tard. En théorie, le processeur exécute les instructions en séquence.



Un compteur temporel ("chronomètre" à la µs au moins) est activable par des ports.

Introduction

Unité centrale de calcul, instructions machine

L'unité centrale de calcul contient quelques douzaines de **registres**, le **compteur ordinal**; des registres spéciaux d'état, et la mémoire cache. Une instruction machine effectue un traitement élémentaire, notamment:

- chargement d'une constante entière codée dans l'instruction -par exemple 4553- dans un registre.
- addition ou multiplication de deux entiers sur 32 ou 64 bits avec résultat dans un registre, et opérandes constants ou en registre.
- lecture d'une case (un mot, un octet, un double-octet, etc...) de la mémoire centrale dans un registre; L'adresse de la case lue peut être indexée par, ou contenue dans, un registre (accès indirect).
- branchement inconditionnel vers une instruction.
- comparaison de deux nombres dans un registre (qui modifie des bits d'état) ou test de signe (comparaison à 0).
- branchement conditionnel vers une instruction.
- écriture d'un registre dans une case mémoire (accès parfois indexé ou indirect).
- appel système.

Introduction

Interruptions

Des événements externes peuvent interrompre le traitement courant:

- fin de temporisation
- accès à une case mémoire interdite
- instruction illicite (code opération unexistant)
- tentative d'exécution d'une instruction priviliégiée** en mode utilisateur
- réception terminée d'un paquet d'octets sur un canal d'entrée
- transmission finalisée d'un paquet d'octets sur un canal de sortie
- implusion électronique sur une broche du processeur

Une **interruption** déroute le calcul (selon un mécanisme câblé et défini) vers une routine exécutée en mode privilégié. Un appel système est semblable à une interruption interne du programme applicatif.

Introduction

Modes d'exécution et instructions priviliégiés

Les programmes applicatifs sont exécutés dans le mode *ordinaire* ou utilisateur. **Dans le mode privilégié** (un bit d'état important), **les instructions machine suivantes sont permises**, mais *interdites* en mode utilisateur :

- *lecture** *d'un port* d'entrée,
- *écriture** *d'un port* de sortie
- (y compris *configuration** *des canaux d'E/S*, et *du mécanisme d'adressage virtuel*)
- *arrêt** du processeur (quand il n'y a rien à faire) en attente d'interruption

Les instructions privilégiées sont marquées par $*$. Les interruptions et les appels systèmes basculent le processeur en mode privilegié.

Introduction

architecture x86-64 - registres généraux

Voir www.amd.com/system/files/TechDocs/24594.pdf

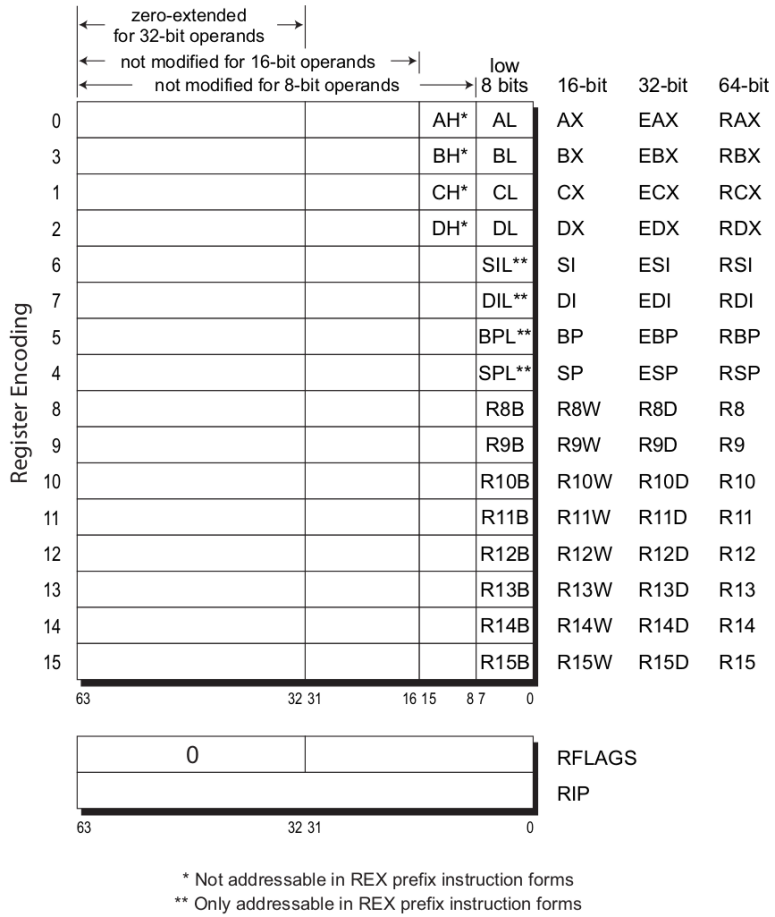


Figure 2-3. General Registers in 64-Bit Mode

Introduction

architecture x86-64 - registres système

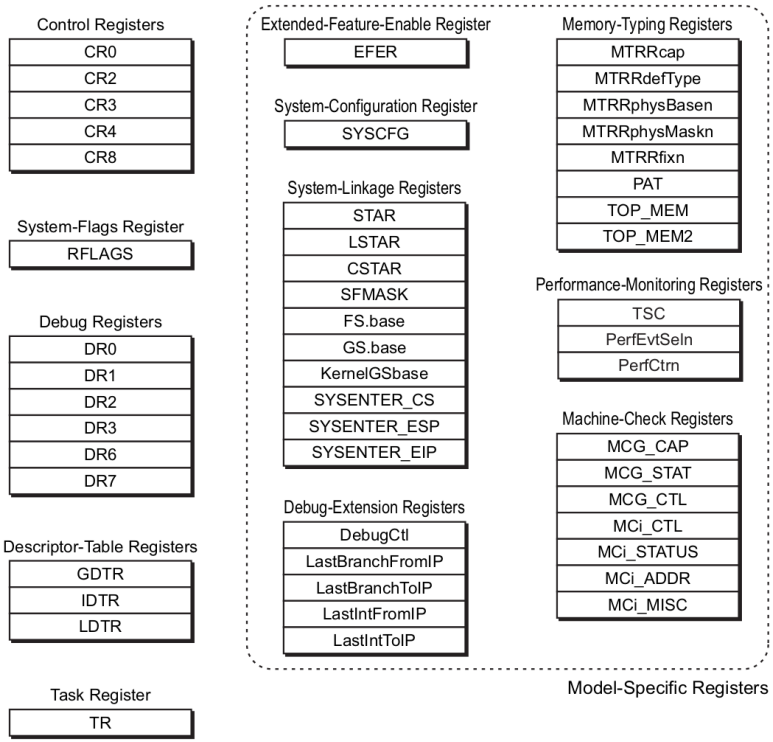


Figure 2-6. System Registers

Par compatibilité, **l'architecture x86-64 est *très* complexe** (documentée en des milliers de pages). Beaucoup d'instructions machines sont "obsolètes" et inefficaces.

Introduction

Points délicats des architectures matérielles : les processeurs multi-cœurs

En pratique en 2021 il y a plusieurs **cœurs de processeur**, donc un traitement concurrent et parallèle de plusieurs flots d'instructions machine.
Les différents cœurs peuvent partager un cache de niveau 3, et peuvent s'interrompre en mode privilégié. Chacun a ses propres registres, ses bits d'états.
La synchronisation des caches et des cœurs nécessitent des mécanismes matériels spécifiques, et posent des problèmes informatiques intéressants (sémaphores, verrous, etc...).

Introduction

Ordre de grandeur temporels en 2021 pour un ordinateur portable

- fréquence d'horloge: 500 MHz à 5 GHz environ.
- temps d'accès à un registre: nanosecondes ($10^{-9}s$) ou moins.
- temps de calcul d'une instruction élémentaire sans défaut de cache: quelques nano-secondes (mais un cœur de processeur execute quelques instructions en parallèle ou à la chaîne - pipeline) ; plusieurs milliards d'instructions machine exécutées chaque seconde dans chaque cœur !
- temps d'accès au cache primaire: dizaine de nanosecondes.
- temps d'accès à la mémoire centrale: centaines de nanosecondes.
- temps d'accès à une bloc (4koctets) du disque SSD: 100 microsecondes.
- temps d'accès à une bloc (4koctets) du disque rotatif: 10 millisecondes.
- temps de transmission d'un kilo-octet sur le réseau Ethernet au PC voisin : dizaines ou centaines de microsecondes
- temps de transmission d'un kilo-octet par Internet au travers l'Atlantique: dizaines de millisecondes.
- cent à mille interruptions par seconde, parfois plus.

Introduction

Ordre de grandeur spatiaux en 2021 pour un ordinateur portable

- centaines de milliards de transistors.
- 4 à 32 cœurs de processeur.
- 4 Go à 64 Go de mémoire centrale.
- 128 Mo à 4Go de mémoire vidéo (carte graphique).
- cache primaire dans chaque cœur : 100 à 200 Ko (un cache pour les instructions, un autre pour les données)
- cache secondaire: 1 à 2 Mo (partagé par les cœurs)
- cache tertiaire: 8 à 32 Mo (partagé par les cœurs)
- disque: SSD de 256 Go; rotatif de quelques To.
- coût: 500 à 2500€

NB. La location mensuelle (chez OVH) d'un serveur virtuel comme `refpersys.org` coûte peu et on y accède avec `bash` via la commande **`ssh`**.

Introduction

Super-calculateurs sous Linux

Pour le calcul scientifique (prévisions météo, simulation numérique, jumeaux numériques, exploration pétrolière, armements, astrophysique, bioinformatique, ...), pendant des heures ou des mois. Machines coûtant plusieurs dizaines de M€.

- même jeu d'instructions machine (x86-64) que votre ordinateur portable (ou PowerPC)
- coprocesseurs de calcul vectoriel spécialisés (OpenCL, etc...) difficiles à programmer.
- dizaines de milliers de processeurs, chacun avec un téraoctet de mémoire centrale.
- petaoctets de disques.
- consommation éléctrique donc dissipation thermique en megawatts.
- matériel coûteux et spécifique pour leur interconnection (MPI) - bande passante en To/sec.

Vos calculs seront pilotés par des scripts en `bash` et devront persister sur disque (ou dans des bases de données) des résultats intermédiaires. Ce mécanisme de sauvegarde de l'état intermédiaire de calcul doit être pensé à l'avance!
NB: un calcul qui dure des semaines peut être faux! - instabilités numériques possibles

Introduction

terminologie, avantages et défauts du bash

- GNU bash est inspiré par le "Bourne shell" (1977), c'est (comme GNU Linux) un **logiciel libre** (que vous pouvez étudier et améliorer; voir ou adhérer à l'APRIL)
- un programme en bash est un **script**
- bash est une variante d'une norme POSIX
- le langage de commande bash est pratique mais peu efficace:
  - presque toutes les étapes élémentaires d'un script bash lancent un processus séparé - donc lenteur des scripts
  - pas de vérification semi-automatique possible. Voir l'exposé de Yann Regis-Gianas à FOSDEM2018 Parsing Posix [S]hell
  - facilite la combinaison de programmes qu'on espère robustes
  - les scripts bash sont souvent peu, mais toujours, lisibles

Introduction

Conseils pratiques

- nommez vos fichiers *sans espaces* (mais avec un blanc souligné _); **évitez** y **les caractères étranges** é ou * ou \ ou ° ou ? ou § ou @ ou : ou %.
- **soignez le choix des noms** (du script, et des identifiants qui y apparaissent).
- par **convention**, un script bash a un nom se terminant par .sh ou .bash, par exemple gros-calcul.bash ou Supprimer_fichiers_inutiles.sh
- **évitez les gros scripts** (sauf quand ils sont générés automatiquement) de plus de 100 lignes; et **documentez vos scripts**
- dans *certains cas* on gagne à **recoder un script** bash **dans un autre langage de programmation** (comme GNU guile, Ocaml, Python, Go ou en *C*) :
  - pour qu'il soit plus rapide
  - par souci de lisibilité et de facilité de maintenance
- en interactif, il existe d'autres "shells", par exemple zsh ou es ou fish
- "shell" autonome de dépannage /bin/sash : Standalone shell par D.Bell qui contient des utilitaires indispensables pour réparer un Linux "cassé".

Introduction

Un premier exemple de commande en GNU bash

Objectif: Déterminer (sous Debian ou Ubuntu) la taille par défaut des feuilles de papier imprimées ou générées au format PDF. En France, c'est le format A4.

Exemple de commande

`/bin/cat /etc/papersize` → a4

Remarques:

- sur un système bien configuré, on aurait pu taper `cat /etc/papersize`
- recommencer et jouer avec la touche *TABulation* en cours de saisie et avec *Ctrl-T*, *Ctrl-A* et *Ctrl-E*
- essayer aussi `cat --version` et `cat --help`
- sur un système mal configuré, on peut avoir des surprises désagréables.
- essayer aussi `cat /dev/null`. Il ne doit "rien" se passer. Voir cat(1) et null(4) pour en savoir plus.
- essayer aussi `cat /etc/machin-chose` (un fichier inexistant)

le système d'exploitation Linux

rôles d'un système d'exploitation

D'après wikipedia:

Definition

système d'exploitation : (anglais "operating system") c'est un ensemble de programmes qui dirige l'utilisation des ressources d'un ordinateur par des logiciels applicatifs

Sur un système Linux, **le programme central est le noyau Linux** dont le code source est en `kernel.org`. **C'est le *seul* programme qui tourne en mode superviseur** sur les cœurs de votre processeur (donc accède au matériel). Il doit être **robuste**. Linux est dérivé et inspiré d'Unix (1970).

août 2021: le dernier noyau stable est Linux 5.13.12 : **20 millions de lignes de code** - MLOC (surtout en C), avec plus de 1500 contributeurs et **revue par les pairs**.

le système d'exploitation Linux

programmes applicatifs

Sans programmes applicatifs, le noyau Linux ne sert à rien.
Le projet GNU fournit des applications en logiciel libre - dont le compilateur GCC (6,4 millions de lignes).
Il faut une distribution Linux pour compiler ou contribuer au noyau Linux, par exemple Debian ou Ubuntu ou RedHat

Il y a plusieurs sortes de programmes applicatifs:

- les **logiciels serveurs**; ils tournent longtemps et sont utilisés par d'autres logiciels: un gestionnaire de base de données comme PostGreSQL, un système de fenêtrage graphique Xorg ou Wayland[1], un serveur Web comme lighttpd ou ocsigen ...
- les **logiciels clients**: ils utilisent les services fournis par le noyau et par des logiciels serveurs. Par exemple un éditeur de code comme GNU emacs ou le navigateur Mozilla FireFox
- les **logiciels simples** n'utilisent que les services fournis par le noyau. Par exemple, le compilateur GCC ou l'interprète GNU `bash` ou les utilitaires de fichiers

[1] Le système de fenêtrage est le seul logiciel accédant *directement* à la souris, au clavier, et à l'écran.

le système d'exploitation Linux

la philosophie originelle d'Unix

Au XX$^{\text{ème}}$ siècle (sur des machines avec peu de mémoire), mais malmenée depuis...

- un programme fait une seule chose, mais "bien" (Keep It Simple, Stupid)
- favoriser des petits programmes réutilisables
- favoriser la combinaison de programmes existants
- documenter les formats de données
- favoriser des fichiers et des données textuels
- respecter le développeur
- logiciel libre et documenté
- génération de code par des programmes, méta-programmation
- étendre Unix avec un système Unix

Aujourd'hui, on a des programmes monstrueux!

le système d'exploitation Linux

fichiers et processus

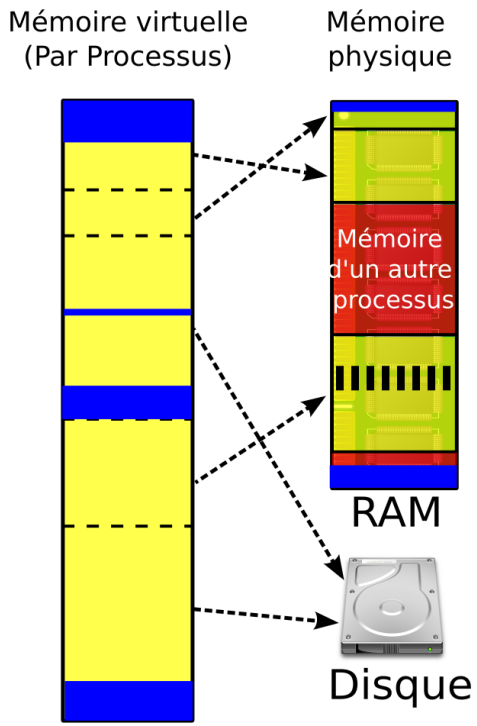
Le noyau fournit deux **abstractions** importantes aux logiciels applicatifs: **les fichiers** et **les processus**. Par exemple, le fichier exécutable `/bin/bash` (au format ELF) contient le programme binaire GNU bash et souvent sera executé par *plusieurs* processus, chacun interprétant un script.

Un **fichier** est une séquence d'octets, souvent representée par une structure organisée de blocs sur le disque. Certains fichiers sont juste des flots d'octets. Un **répertoire** est un cas particulier de fichier, connu du noyau, associant des noms à des fichiers. Un **lien symbolique** est un autre cas particulier de fichier, contenant un nom (chemin complet) de fichier et permettant une indirection sur les noms. Un même fichier peut n' avoir aucun nom (s'il a été supprimé de tous les répertoires mais reste utilisé par un processus), ou plusieurs noms dans des répertoires distincts.

Un **processus** **est un programme *en cours d'exécution***. Il accède aux fichiers par des **descripteurs de fichiers ouverts**. Chaque processus a son **espace d'adressage** propre (mémoire virtuelle).

le système d'exploitation Linux

Mémoire virtuelle et espace d'addressage d'un processus



D'après Wikipedia - simplifié

Commandes bash à essayer: `/bin/cat /proc/self/maps`
et `/bin/cat /proc/$$/maps` ; voir `bash(1)`
et `cat(1)` et `proc(5)`

le système d'exploitation Linux

bibliothèques logicielles et greffons

Une **bibliothèque logicielle** ("software library") est un paquet de code conçu et destiné à être réutilisée dans plusieurs applications. Elle peut être distribuée sous forme source et/ou sous forme binaire. Sa documentation définit comment l'utiliser.

Une **bibliothèque binaire** est **partagée** ("shared library") quand des segments de code de son fichier sont utilisés par plusieurs processus.

Certains logiciels (par exemple le navigateur Firefox, et les GNU `bash` récents) peuvent à l'exécution charger[2] des bibliothèques binaires partagées qui sont des **greffons** ("plug-ins") étendant les fonctionalités du logiciel de base[3].

[2] Le chargement d'un greffon se fait les fonctions `dlopen` et `dlsym` durant l'exécution et modifie la mémoire virtuelle du processus chargeur.

[3] Certains noyaux Linux acceptent des "modules" chargés dynamiquement par `modprobe`. Ces modules sont l'équivalent pour le noyau des greffons pour les applications.

le système d'exploitation Linux

Linux est multi-tâches et multi-utilisateurs

Linux est **multi-tâches** car capable d'exécuter plusieurs processus[4] quasi-simultanément. Le noyau se charge de répartir les processus actifs sur les cœurs. L'**ordonnanceur** ("scheduler") du noyau donnera la main à tour de rôle à chaque processus actif, et l'interrompera périodiquement (quantum de temps en millisecondes) et passera la main à un autre processus. Le noyau Linux donne à l'ensemble des applications l'**illusion** de beaucoup de cœurs virtuels, chacun exécutant un processus.

Linux est **multi-utilisateurs** car il contrôle l'accès aux ressources en fonction des utilisateurs. Un utilisateur est identifié dans le noyau par son numéro[5], le **uid** ("user id", identifiant d'utilisateur). **Chaque processus et chaque fichier appartient à un utilisateur** et connait son *uid*. Plusieurs personnes utilisatrices d'un serveur de calcul peuvent ainsi exécuter leurs programmes sans interférence possible entre leurs processus. **Le noyau contrôle et restreint les communications entre processus**.

[4] Sur la machine de bureau où sont tapés ces transparents, il y a 730 processus, dont quelques uns sont actifs. Sur de coûteux serveurs de calcul, on peut avoir des dizaines de milliers de processus, dont plusieurs centaines sont actifs.

[5] Les utilisateurs sont décrits dans `/etc/passwd`, voir `passwd(5)`.

le système d'exploitation Linux

Création des processus

Chaque processus connu du noyau est identifié par un numéro unique positif, son **pid** identifiant de processus. **Un processus est toujours créé par un autre, son processus père**, sauf le processus initial `/sbin/init` de pid 1, démarré par le noyau[6], et quelques autres (auto-detection par le noyau de l'insertion d'une clef USB).

La primitive de création d'un processus est l'appel système fork documenté en `fork(2)` **qui duplique le processus courant** et renvoie un *pid*:

- dans le processus père, `fork` renvoie le pid du fils nouvellement créé
- *simultanément* **dans le processus fils**, `fork` renvoie 0, et **l'espace d'addressage a été dupliqué** ("copy on write").
- en cas d'échec, `fork` renvoie -1 (dans le père)
- **il n'y a pas d'autre différence** (outre leur pid) **entre processus père et fils**.

Les processus père et fils sont des clones quasi-parfaits. Souvent, le processus fils exécutera rapidement un autre programme par l'appel système `execve(2)`

[6] Avant 2010, cet `/sbin/init` était un script GNU `bash`; mais c'est maintenant systemd sur la plupart des distributions.

le système d'exploitation Linux

Exécution d'un nouveau programme

Le GNU `bash` va utiliser `fork(2)` pour presque chaque commande executée[7]. La plupart des commandes vont démarrer un nouveau processus. Cette exécution a lieu dans le processus fils (de votre processus `bash`) par l'appel système `execve(2)` qui:

- reçoit comme arguments:
  - le **chemin du programme à exécuter**, e.g. `/usr/local/bin/refpersys`
  - un tableau de chaînes de caractères, les **arguments du programme**
  - un tableau d'**affectation de variables d'environnement**, dont `HOME`. Sur mon système, j'ai `HOME=/home/basilesta/` et des dizaines d'autres variables d'environnement, dont `LANG=en_US.utf8` et `DISPLAY=:0.0`
- réinitialise l'espace d'addressage du processus suivant les indications contenues dans le programme binaire à exécuter.
- réinitialise des compteurs de temps, supprime les traitements des signaux, etc...

[7] Cet appel système `fork` est rapide; dans des cas favorables une centaine de microsecondes.

le système d'exploitation Linux

Terminaison d'un processus

Un processus (qui exécute un programme) peut se terminer par:

- l'appel système exit(2) qui transmet un code d'erreur entre 0 et 127; par convention le code 0 signifie le succès du programme, et les autres codes indiquent des échecs.
- une anomalie à l'exécution (par exemple l'accès à une adresse illicite, l'exécution d'une instruction machine privilégiée ou inexistante) se manifestant par un signal.
- le dépassement de certaines ressources (de temps, d'espace d'adressage, d'espace disque, voir `setrlimit(2)`, etc...)
- la réception d'un signal envoyé (avec `kill(2)`...) par un autre processus.

Avec GNU `bash` vous pouvez tuer votre processus de pid 1234 par la commande `kill 1234`

Le noyau récupère et gère les ressources du processus défunt. Certains sont zombies.

le système d'exploitation Linux

Les signaux

C'est une forme de communication ou notification asynchrone frustre entre processus (analogue aux interruptions du matériel). Un processus peut envoyer un signal à un autre processus par l'appel système `kill(2)`. Le noyau envoie des signaux.

Un processus reçoit un signal du noyau en cas d'anomalie (accès à une adresse invalide, division par 0) ou de dépassement d'un chronomètre. Voir `alarm(2)`.
Il peut demander à recevoir un signal quand des données arrivent (par Ethernet) ou quand le courant manque.

La primitive de GNU bash pour traiter les signaux est `trap`, et pour envoyer un signal `kill`

Il existe des douzaines de signaux différents: `SIGINT`, `SIGTERM`, `SIGKILL` (impossible à éviter, mortel), `SIGCHLD`, `SIGALARM`, etc.... Les lister par **`kill -l`**.

Voir aussi `signal(7)` et surtout `signal-safety(7)`

Pour tuer un processus serveur proprement de pid 1234, `kill -TERM 1234` (s'il est bien programmé, il terminera proprement). Pour le tuer salement, `kill -KILL 1234`

le système d'exploitation Linux

Le plantage d'un processus: fichier core et les limites d'un processus

Dans certains cas, un processus plante et le noyau peut vider son état applicatif dans un fichier nommé souvent `core`[8]. Détails dans `core(5)`

Ce plantage est notifié par un signal émis par le noyau.

Le fichier `core` a un format bien défini, et est analysable par un débogueur comme `gdb(1)` ou d'autres programmes.

Le fichier `core` contient l'image instantanée de l'espace d'addressage du processus défunt.

Un processus peut limiter ou interdire le vidage d'un fichier `core` par l'appel système setrlimit(2) qui peut limiter aussi : le temps de calcul, l'espace d'addressage, la mémoire physique ou virtuelle utilisée, le nombre de descripteurs de fichiers, la taille des fichiers, le nombre de processus créés, etc...

GNU `bash` offre la primitive **`ulimit`** pour interfacer cet appel système.

[8] Historiquement (en 1970), la mémoire centrale (moins d'un Mo!) du PDP 11 ou de l'IBM 360 était à tores de ferrite, en anglais "ferrite *core* memory".

le système d'exploitation Linux

Typologie des signaux

Pour *certains* signaux (marqués *), un processus peut installer une action (routine traitant ce signal) par l'appel système `sigaction(2)`. Certains signaux sont ignorés, d'autres (marqués °) provoquent un vidage `core` avec l'arrêt du processus, et certains (marqués /) terminent le processus si ils n'ont pas de traiteur correspondant.

Le traitement robuste d'un signal est *délicat et difficile* : il peut être reçu à n'importe quel moment (y compris au milieu d'un appel de fonction) ! En principe, pas de `printf(3)` dans un traiteur de signal. Souvent, positionnement d'un drapeau de type `volatile sigatomic_t` à tester régulièrement ailleurs dans un exécutable codé en C.

le système d'exploitation Linux

Signaux importants

* traiteur possible ou souhaitable; ° vidage `core`; / terminaison sans traiteur.

nom du signal	*origine*	*descriptif et usage*
`SIGSTOP`	processus	stoppe le processus
`SIGCONT*`	processus	continue et reprend un processus stoppé par `SIGSTOP`
`SIGFPE°*`	interne	division par 0 ou erreur arithmétique
`SIGILL°*`	interne	instruction machine interdite
`SIGSYS°*`	interne	mauvais appel système
`SIGINT°*`	clavier *Ctrl* C	interruption par le clavier
`SIGCHD*`	interne	processus fils terminé
`SIGHUP*`	interne	déconnexion du terminal, etc...
`SIGIO*/`	interne	entrée/sortie possible...
`SIGKILL°`	interne	tue le processus recepteur; signal intraitable
`SIGSEGV*°`	interne	addresse mémoire illicite
`SIGTERM*/`	processus	terminaison du processus (traités dans logiciels serveurs)
`SIGXCPU*/`	interne	dépassement de la limite de temps de calcul
`SIGXFZ*/`	interne	dépassement de la limite de taille d'un fichier
`SIGALRM*/`	interne	dépassement d'un chronomètre réel
`SIGVTALRM*/`	interne	dépassement d'un chronomètre de temps calcul
etc....		

Voir `signal(7)` et `signal-safety(7)`

le système d'exploitation Linux

La variable d'environnement PATH

Cette variable contient une liste *ordonnée* de répertoires (séparés par `:`) pouvant contenir des programmes et des scripts. On l'interroge par la commande `echo $PATH` qui donne chez moi la liste:
`/home/basilesta/bin:/usr/local/bin:/usr/bin:/bin:/sbin`

NB: Autrefois `/usr/` a été (en 1975) un gros disque pour les utilisateurs d'Unix et leur programmes, d'où le nom...

Le `bash`, les autres shells, comme les fonctions `execv(3)` et `execl(3)` utilisent la variable `PATH`.
Ainsi, si je tape `date` comme commande pour `bash`, ce shell cherchera à exécuter dans l'ordre les programmes ou scripts en `/home/basilesta/bin/date` (absent), puis en `/usr/local/bin/date` (absent) enfin en `/usr/bin/date` (présent) qui sera (après le `fork` habituel) executé par `bash` à l'aide de `execve(2)`

le système d'exploitation Linux

Le superutilisateur

Pour les processus dont le uid est 0, les appels systèmes sont moins contrôlés.

Par convention, le super-utilisateur `root` dont le uid est 0 a tous les droits (donc une grande responsabilité).

Le mécanisme de changement d'utilisateur s'appelle **setuid** - c'est une propriété de certains fichiers exécutables, permettant les appels systèmes dangereux `setuid(2)` et `seteuid(2)`

Les programmes `/usr/bin/sudo` et `/bin/su` permettent de passer en `root` et les executables correspondants sont *setuid*.

Conseil: **évitez d'utiliser *root* longtemps**, et utilisez avec crainte et attention la machinerie *setuid* dans vos programmes (en C). Voir la commande `chmod(1)`