Shell

Gestion des processus et entrées-sorties

Ensimag, 2A, édition 2022-2023

29 septembre 2023

TP 2 : le shell

Opérations sur les processus
- Processus
- Création de processus
- Recouvrement de programme
- Attente de la terminaison

les IO
- Rappel sur les processus
- Le shell
- Le pipe

Outils

Shell

Le but est de faire un *shell* de commandes (interpréteur simpliste de commandes fourni)

- lancer les programmes demandés avec leurs arguments,
- gérer l'attente éventuelle et la terminaison des processus.

Il y a aussi : les redirections d'entrées-sorties (`<`, `>`, `|`) et quelques variantes (libreadlines, signaux, joker, libcurses, Ctrl-Z/fg/bg, etc.).

Le sujet est sur la page du cours ! ! !

Attention : le sujet est sur la page du cours !

De l'aide ?

- `man fork`
- `man execvp`
- `man 2 wait`
- ...
- En cas d'ambiguité :
  - `whatis` *commande* puis
  - `man` *N commande*
  - Exemple : `man 2 open` pour le open du C, `man 3 open` pour le open de perl ...)

Processus

Definition (Qu'est-ce qu'un processus ?)

Processus

Definition (Qu'est-ce qu'un processus ?)

Un processus est un *programme* en *exécution*

Programme

Qu'est-ce qu'un programme ?

Programme

Qu'est-ce qu'un programme ?

- du code : des séquences d'instructions regroupées en fonctions
- des données : des valeurs constantes, des variables globales initialisées, la taille totale des variables globales non initialisées, une liste des dépendances dynamiques à charger à l'exécution (librairie C, etc.)

Exécution

Qu'est-ce que l'exécution d'un programme ?

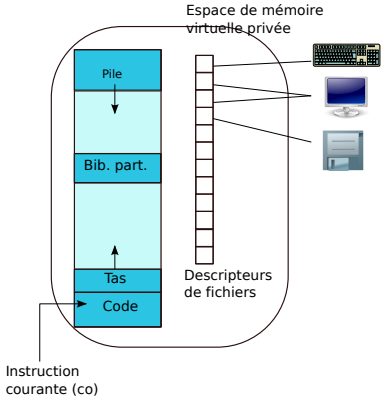
- 1 ou plusieurs threads, les entités exécutant les fonctions
- des registres mémorisant **par cœur** une partie de l'état du thread en train d'être exécuté par le cœur :
  - **le compteur ordinal** stockant l'instruction courante du thread exécuté
  - **le registre d'état** stockant un bitfield (tableau de booléens : division par 0, masquage des interruptions, résultats de tests de branchements, etc.)
  - **le pointeur de pile** stockant l'adresse courante de la pile du thread
  - des registres généraux de calculs
- ... (suite slide suivant)

Exécution (2/2)

Qu'est-ce que l'exécution d'un programme ?

- de la mémoire (RAM) stockant 1 pile par thread (les variables locales des fonctions en cours) et les parties utiles du programme (code et données)
- la sauvegarde en RAM de l'état des registres des threads qui ne sont pas élus : l'état des threads en attente (bloqué sur I/O, ou attendant un cœur disponible)
- des entrées-sorties : la liste des fichiers ouverts, des tampons gardant en RAM des morceaux de fichiers stockés sur disque ou sur le réseau, etc.

Un processus dans un OS moderne



Processus (avec 1 seul thread)

- Un espace de mémoire virtuelle privée
  - le code,
  - la pile : variables locales des fonctions
  - le tas : malloc/free
  - des segments de mémoire partagée : bibliothèques de fonctions partagées
- Le compteur ordinal (adresse de l'instruction courante du thread)
- Des registres
- Des descripteurs de fichiers : E/S vers fichiers, écran, clavier, réseau, etc.

Création de processus sous UNIX

Création par l'appel système `fork()`

Mais que fait `fork()` ? Comment savoir ?

Création de processus sous UNIX

Création par l'appel système `fork()`

Mais que fait `fork()` ? Comment savoir ?

```
FORK(2)          Linux Programmer's Manual          FORK(2)
NAME
       fork - create a child process
SYNOPSIS
       #include <unistd.h>
       pid_t fork(void);
DESCRIPTION
       fork()  creates  a  new  process by duplicating the
       calling process.  The new process, referred  to  as
       the  child,  is  an  exact duplicate of the calling
       process, referred to as the parent, except for  the
       following points:
```


Création de processus sous UNIX

`int fork()` : création par copie

La création se fait par copie à l'identique du processus qui appelle la fonction `fork`.

Création de processus sous UNIX

`int fork()` : création par copie

La création se fait par copie à l'identique du processus qui appelle la fonction `fork`.

La règle : copie *TOUT, pour tous les threads*, états des registres, état de la mémoire (programme, données, les piles, tas, la plupart des segments partagées) et les entrées sorties.

Création de processus sous UNIX

`int fork()` : création par copie

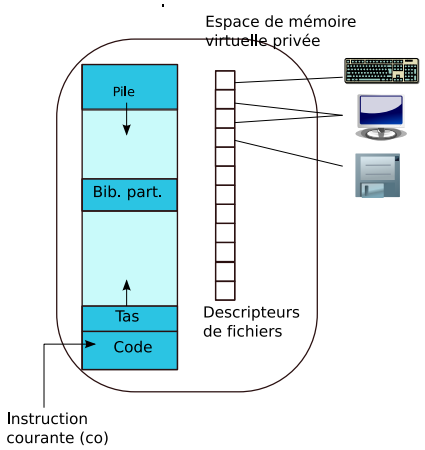
La création se fait par copie à l'identique du processus qui appelle la fonction `fork`.

La règle : copie *TOUT, pour tous les threads*, états des registres, état de la mémoire (programme, données, les piles, tas, la plupart des segments partagées) et les entrées sorties.

Les exceptions : *la valeur de retour de* `fork` *(0 dans le fils, PID du fils dans le père)*, l'identification du processus (numéro unique, PID), l'identification des threads (numéro unique, TID), les verrous, les statistiques (`getrusage()`), les alarmes (`setitimer()`)

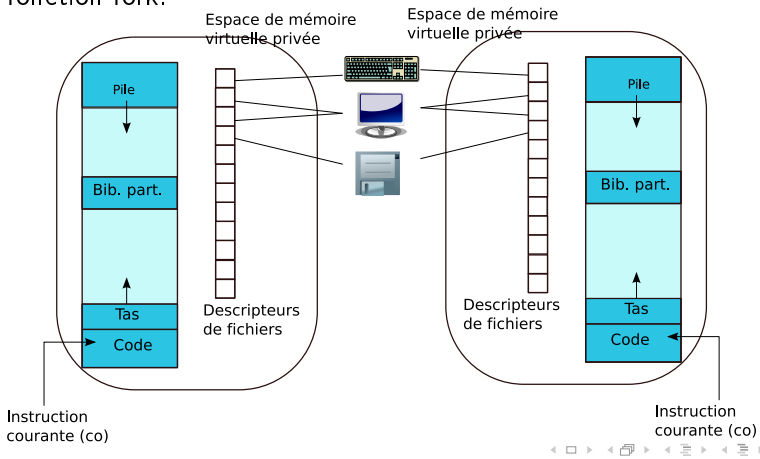
Création de processus avec fork

Le thread du processus père commence l'exécution de fork.



Création de processus avec fork

Le thread du processus père commence l'exécution de fork. Les deux threads des deux processus terminent leurs exécutions de la fonction fork.



Création d'un processus avec fork

```
pid_t pid = fork();
switch(pid) {
case -1:
  perror("fork:"); break;
case 0:
  printf("Ahhh !!!!!\n"); break;
default:
  printf("%d, je suis ton père\n", pid);
  break;
}
```


Créations multiples

Boucle simple

Que fait le code suivant ?

```
for(i=0; i< n; i++) {
  fork();
}
```

Créations multiples

Boucle simple

Que fait le code suivant ?

```
for(i=0; i< n; i++) {
  fork();
}
```

Il crée 2^n processus ! (CAUTION : fork bomb ! Rappel aux imprudents : les processus ont un propriétaire :-) !)

Créations multiples

La bonne solution : tester la valeur de retour de fork

```
for(i=0; i< n; i++) {
  int pid = fork();
  if (! pid) // pid == 0, on est dans le fils, on break
    break;
}
```


Filiation

ou bien

```
for(i=0; i< n; i++) {
  int pid = fork();
  if (pid) // pid != 0, on est dans le père, on break
    break;
}
```

Quelle est la différence entre les deux codes ?

Quel impact sur la filiation ?

One call to rule them all

Sous Linux, fork est implanté en utilisant l'appel système `clone`, utilisé aussi pour créer des threads, ou dans certaines techniques de virtualisation.

```
CLONE(2)                   Linux Programmer's Manual                  CLONE(2)
NAME
       clone, __clone2, clone3 - create a child process
SYNOPSIS
       int clone(int (*fn)(void *), void *stack, int flags, void *arg, ...
                 /* pid_t *parent_tid, void *tls, pid_t *child_tid */ );
       long clone3(struct clone_args *cl_args, size_t size);
DESCRIPTION
       These  system calls create a new ("child") process, in a manner similar
       to fork(2).

       By contrast with fork(2), these system calls provide more precise  con-
       trol over what pieces of execution context are shared between the call-
       ing process and the child process.  For  example,  using  these  system
       calls,  the  caller  can control whether or not the two processes share
       the virtual address space, the table of file descriptors, and the table
       of  signal  handlers.   These  system  calls  also  allow the new child
       process to be placed in separate namespaces(7).
```

Le recouvrement

`execvp, execve, execlp, execle` : le recouvrement

L'appel à exec remplace le programme (code + données) du processus par un autre programme. Puis le processus recommence au début du nouveau programme (main).

Il est souvent utilisé après un fork.

TP 2 : le shell | Opérations sur les processus | les IO | Outils

Le recouvrement

`execvp, execve, execlp, execle` : le recouvrement

L'appel à exec remplace le programme (code + données) du processus par un autre programme. Puis le processus recommence au début du nouveau programme (main).

Il est souvent utilisé après un fork.

Pas de création !

Exec ne crée pas un nouveau processus ! Il remplace le programme d'un processus en cours de route. Il termine donc aussi tous les autres threads puisqu'ils n'ont plus ni code, ni données !

TP 2 : le shell | Opérations sur les processus | les IO | Outils

Le recouvrement

`execvp, execve, execlp, execle` : le recouvrement

L'appel à exec remplace le programme (code + données) du processus par un autre programme. Puis le processus recommence au début du nouveau programme (main).

Il est souvent utilisé après un fork.

Pas de création !

Exec ne crée pas un nouveau processus ! Il remplace le programme d'un processus en cours de route. Il termine donc aussi tous les autres threads puisqu'ils n'ont plus ni code, ni données !

Ne revient pas !

Un appel réussi à exec ne revient jamais (sauf erreur) !

TP 2 : le shell | Opérations sur les processus | les IO | Outils

L'attente

`wait, waitpid` : l'attente de la terminaison

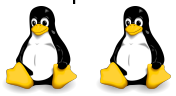
Les fonctions `pid_t wait(int *status)` ou `pid_t waitpid(pid_t pid, int *status, int options)` permettent à un processus parent d'attendre la fin d'un de ses fils directs. Le processus père récupère la valeur entière donnée en argument à `exit(int)` ou en retour du `main`.

En résumé

- `fork` permet la création d'un nouveau processus par copie

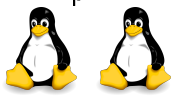
En résumé

- `fork` permet la création d'un nouveau processus par copie



En résumé

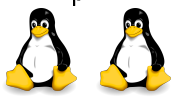
- `fork` permet la création d'un nouveau processus par copie



- exec change le programme exécuté par un processus

En résumé

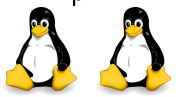
- `fork` permet la création d'un nouveau processus par copie



- exec change le programme exécuté par un processus

En résumé

- `fork` permet la création d'un nouveau processus par copie



- `exec` change le programme exécuté par un processus

- `wait` : le processus père attend la fin de son fils.

La création de processus UNIX

Création en deux étapes

- La création par copie à l'identique (fork),
- Le remplacement du programme en cours (exec).

Pourquoi séparer les opérations ?

La création de processus UNIX

Création en deux étapes

- La création par copie à l'identique (fork),
- Le remplacement du programme en cours (exec).

Pourquoi séparer les opérations ?

Pour faire des modifications sur les entrées-sorties

La création de processus UNIX

Création en deux étapes

- La création par copie à l'identique (fork),
- Le remplacement du programme en cours (exec).

Pourquoi séparer les opérations ?

Pour faire des modifications sur les entrées-sorties

1. Le nouveau processus créé par le fork a les mêmes entrées-sorties que son père,

3. l'execvp ne change pas les entrées-sorties

La création de processus UNIX

Création en deux étapes

- La création par copie à l'identique (fork),
- Le remplacement du programme en cours (exec).

Pourquoi séparer les opérations ?

Pour faire des modifications sur les entrées-sorties

1. Le nouveau processus créé par le fork a les mêmes entrées-sorties que son père,
2. **FAIRE LES CHANGEMENTS DES I/O ICI**
3. l'execvp ne change pas les entrées-sorties

La création de processus Windows

Une seule fonction, est-ce plus facile ?

Dans l'API Windows, la création processus se fait avec une seule fonction. Elle a donc besoin de nombreux paramètres pour, finalement, ne traiter uniquement que les cas de bases prévus par MS.

CreateProcess

```
BOOL WINAPI CreateProcess(
_In_opt_     LPCTSTR lpApplicationName,
_Inout_opt_  LPTSTR lpCommandLine,
_In_opt_     LPSECURITY_ATTRIBUTES lpProcessAttributes,
_In_opt_     LPSECURITY_ATTRIBUTES lpThreadAttributes,
_In_         BOOL bInheritHandles,
_In_         DWORD dwCreationFlags,
_In_opt_     LPVOID lpEnvironment,
_In_opt_     LPCTSTR lpCurrentDirectory,
_In_         LPSTARTUPINFO lpStartupInfo,
_Out_        LPPROCESS_INFORMATION lpProcessInformation
);
```


Les descripteurs de fichiers

Pour un processus, (presque) toutes les entrées-sorties passent par des descripteurs de fichiers. Un descripteur est accédé par son numéro. Toutes les opérations utilisent ces numéros. Les descripteurs sont partagés par tous les threads.

Quelles sont les opérations classiques sur les entrées-sorties ?

TP 2 : le shell | Opérations sur les processus | les IO | Outils

Les descripteurs de fichiers

Pour un processus, (presque) toutes les entrées-sorties passent par des descripteurs de fichiers. Un descripteur est accédé par son numéro. Toutes les opérations utilisent ces numéros. Les descripteurs sont partagés par tous les threads.

Quelles sont les opérations classiques sur les entrées-sorties ?

- l'ouverture (int `open(...)`, int socket(...), pipe(...)) qui renvoie le numéro du descripteur ouvert

Les descripteurs de fichiers

Pour un processus, (presque) toutes les entrées-sorties passent par des descripteurs de fichiers. Un descripteur est accédé par son numéro. Toutes les opérations utilisent ces numéros. Les descripteurs sont partagés par tous les threads.

Quelles sont les opérations classiques sur les entrées-sorties ?

- l'ouverture (int `open(...)`, int socket(...), pipe(...)) qui renvoie le numéro du descripteur ouvert
- la lecture (read(int fd, ...), recv(int fd, ...)),

TP 2 : le shell | Opérations sur les processus | les IO | Outils

Les descripteurs de fichiers

Pour un processus, (presque) toutes les entrées-sorties passent par des descripteurs de fichiers. Un descripteur est accédé par son numéro. Toutes les opérations utilisent ces numéros. Les descripteurs sont partagés par tous les threads.

Quelles sont les opérations classiques sur les entrées-sorties ?

- l'ouverture (int `open(...)`, int socket(...), pipe(...)) qui renvoie le numéro du descripteur ouvert
- la lecture (read(int fd, ...), recv(int fd, ...)),
- l'écriture (write(int fd, ...), send(int fd, ...)),

TP 2 : le shell | Opérations sur les processus | les IO | Outils

Les descripteurs de fichiers

Pour un processus, (presque) toutes les entrées-sorties passent par des descripteurs de fichiers. Un descripteur est accédé par son numéro. Toutes les opérations utilisent ces numéros. Les descripteurs sont partagés par tous les threads.

Quelles sont les opérations classiques sur les entrées-sorties ?

- l'ouverture (int `open(...)`, int socket(...), pipe(...)) qui renvoie le numéro du descripteur ouvert
- la lecture (read(int fd, ...), recv(int fd, ...)),
- l'écriture (write(int fd, ...), send(int fd, ...)),
- la fermeture (close(int fd), shutdown(int fd,...)).

TP 2 : le shell | Opérations sur les processus | les IO | Outils

La gestion des entrées-sorties

Definition (Les entrées-sorties standards)

Par convention, chaque processus s'attend à avoir à son démarrage trois descripteurs d'entrées-sorties ouverts :

- l'entrée standard (stdin), dans le descripteur 0,
- la sortie standard (stdout), dans le descripteur 1,
- la sortie d'erreur standard (stderr), dans le descripteur 2.

Les fonctions des bibliothèques standards

Elles ne s'occupent pas de savoir vers quoi sont ouverts les descripteurs : terminal, fichiers, sockets réseaux, etc.
`printf("toto")` **réalise un** `write(1, "toto", 4)`

Le shell

Le shell peut donc rediriger à la demande les entrées-sorties standards des processus. Par exemple pour `./exemple < le_fichier.txt` :

```
// ouvrir un descripteur vers l'entree-sortie
int fd = open("le_fichier.txt", O_RDONLY);
if (fd == -1) { perror("open: "); exit(EXIT_FAILURE);}
// fermer le descripteur standard et dupliquer
// le descripteur ouvert dans le descripteur standard
dup2(fd, STDIN_FILENO); // STDIN_FILENO == 0
// fermer le descripteur ouvert en double
close(fd);
```


Les tuyaux (pipe)

- Les tuyaux sont des outils de synchronisation de type producteur-consommateur qui connectent la sortie d'un processus avec l'entrée d'un autre.

  `ls -R | egrep '.c$' | less`
- Comme le tuyau est de petite taille (quelques kilos), il synchronise les "vitesses" de production et de consommation : la puissance de calcul est répartie et l'occupation mémoire constante, indépendamment la longueur du flot.
- Un tuyau est un objet anonyme, en conséquence, il n'est connecté qu'avec les processus qui ont un descripteur ouvert sur lui.

`int pipe(int fds[2])`

1. L'appel `int pipe(int fds[2])` crée un tuyau et renvoie les numéros des deux descripteurs vers le tuyau (`fds[0]` pour lire, `fds[1]` pour écrire)
2. ensuite on peut faire des *fork* pour créer des processus connectés au pipe.

Détection de la fin de l'écriture dans un pipe

Une communication par tuyau est terminée quand :

- il est vide,
- aucun processus ne peut écrire dedans (tous les descripteurs en écriture sont maintenant fermés), y compris les descripteurs des processus bloqués en lecture dans le pipe.

`int pipe(int fds[2])`

```
int res;
char *arg1[]={"ls","-R", 0};
char *arg2[]={"egrep","\.c$", 0};
int tuyau[2];

pipe(tuyau);
res = fork()
if(res == 0) { // si on est dans le fils
  dup2(tuyau[0], 0); // lecture de stdin dans le tuyau
  close(tuyau[1]); close(tuyau[0]);
  execvp(arg2[0],arg2); // egrep. Ne retourne jamais
}
dup2(tuyau[1], 1); // ecriture de stdout dans le tuyau
close(tuyau[0]); close(tuyau[1]);
execvp(arg1[0],arg1); // ls. Ne retourne jamais
```

Valgrind

Valgrind est un outil de débogage (et plus) capable de vérifier la pertinence des accès mémoires et indiquer les erreurs :

- débordement de tableaux,
- variable non initialisée,
- réutilisation d'un pointeur déjà libéré, etc.

Il fonctionne en instrumentant le code à l'exécution pour tracer les accès mémoires. Néanmoins, il ne fait pas des miracles.

```
int a = 0;
int tab[2] = {};
int b = 0;
...
tab[2] = 12; // Modifie a ou b ! (c'est valide !)
```


Valgrind

Exemple de code faux

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
  char tampon[256]={}; char *copie;
  scanf("%255s", tampon);
  copie = malloc(strnlen(tampon, 256));
  strncpy(copie, tampon, 256); return 0; }
```

Tampon de taille limitée en mémoire locale

Pourquoi faut-il faire particulièrement attention à la taille des entrées (%255s) ? (Indication : que font `call toto` et `ret` en assembleur x86 ?)

Valgrind

Messages d'erreurs

```
==20137== Memcheck, a memory error detector
==20137== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==20137== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -h for copyrigh
==20137== Command: ./a.out
==20137== Invalid write of size 1
==20137==    at 0x4025DB0: strncpy (mc_replace_strmem.c:329)
==20137==    by 0x80484DE: main (exemple_valgrind.c:10)
==20137==  Address 0x41a202c is 0 bytes after a block of size 4 alloc'd
==20137==    at 0x4024C4C: malloc (vg_replace_malloc.c:195)
==20137==    by 0x80484B8: main (exemple_valgrind.c:9)
==20137==
==20137== Invalid write of size 1
==20137==    at 0x4025DBD: strncpy (mc_replace_strmem.c:329)
==20137==    by 0x80484DE: main (exemple_valgrind.c:10)
==20137==  Address 0x41a202e is 2 bytes after a block of size 4 alloc'd
==20137==    at 0x4024C4C: malloc (vg_replace_malloc.c:195)
==20137==    by 0x80484B8: main (exemple_valgrind.c:9)
```


Valgrind

Messages d'erreurs

```
==20137==
==20137==
==20137== HEAP SUMMARY:
==20137==     in use at exit: 4 bytes in 1 blocks
==20137==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==20137==
==20137== LEAK SUMMARY:
==20137==    definitely lost: 4 bytes in 1 blocks
==20137==    indirectly lost: 0 bytes in 0 blocks
==20137==      possibly lost: 0 bytes in 0 blocks
==20137==    still reachable: 0 bytes in 0 blocks
==20137==         suppressed: 0 bytes in 0 blocks
==20137== Rerun with --leak-check=full to see details of leaked memory
==20137==
==20137== For counts of detected and suppressed errors, rerun with: -v
==20137== ERROR SUMMARY: 252 errors from 2 contexts (suppressed: 11 fr
```

valgrind et gdb

Valgrind vérifie vos allocations et initialisations de Variables

L'utilisation systématique de valgrind permet de détecter certains bugs dès leurs introductions (dans le TP shell : copie ratée de chaîne de caractère, paramètres d'appels systèmes manquants).

```
valgrind ./monshell
```

gdb permet de tester l'état d'un processus

On peut attacher gdb à un processus déjà en exécution et inspecter son état.

```
gdb ./monshell 1234
```

pour un processus de PID 1234

valgrind et gdb

On peut même faire les deux en même temps

Valgrind propose à gdb de le piloter lors d'un débogage.

1. lancer le processus avec valgrind et le stopper au lancement
   ```
   valgrind --vgdb=yes --vgdb-error=0 ./monshell
   ```
2. lancer gdb
   ```
   gdb ./monshell
   ```
3. dans gdb, se connecter à valgrind
   ```
   target remote | vgdb
   ```
4. dans gdb, mettre des breakpoints et continuer
   ```
   break la_fonction_a_deboguer
   continue
   ```


gdb et le processus fils

Continuer gdb dans le fils après le fork

Lors d'un fork, il est possible de demander à gdb de continuer dans le fils au lieu de rester dans le père (le défaut).

```
# dans la console de gdb
set follow-fork-mode child
show follow-fork-mode # pour vérifier
```


strace (unix only, dtrace on MacOS)

Il est possible de lire la liste de tous les appels systèmes d'un processus et leurs paramètres.

```
$ strace bash -c 'ls > /dev/null' 2>&1 | wc -l
183
$ strace bash -c 'ls > /dev/null' 2>&1 | grep clone
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_
↪  CHILD_SETTID|SIGCHLD, child_tidptr=0x7fdd3fe09a10) =
↪  17620
$ strace bash -c 'ls | grep toto' 2>&1 | grep -E
↪  'clone|pipe'
pipe([3, 4])                            = 0
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_
↪  CHILD_SETTID|SIGCHLD, child_tidptr=0x7f0b86f2ba10) =
↪  17978
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_
↪  CHILD_SETTID|SIGCHLD, child_tidptr=0x7f0b86f2ba10) =
↪  17979
$
```

perf (linux only) : savoir où vous passez votre temps dans le noyau

perf permet de tracer finement les fonctions du noyau utilisées à un instant donné, ou utilisées par un processus particulier.

```
# top, pour le noyau, par échantillonage
perf top
# tracer ce que le processus PID fait pendant 10 secondes
perf record -p PID sleep 10
# afficher le résultat de la trace
perf report
# et plein d'autres choses (défaut de cache, etc.)
```