R1.04 - Systèmes - TP 3

Processus

Programme, Commande, Application

Programme

Quand on parle de *programme*, on peut parler de deux choses :

- Un *programme source*, qui est écrit dans un langage de programmation comme le *C* par exemple, et qui est compréhensible par un être humain mais pas par l'ordinateur.
- Un *programme binaire*, qui est généralement le résultat de la compilation du programme source en quelque chose de compréhensible par l'ordinateur, mais plus du tout par un être humain (à part peut-être certains barbus oubliés dans un bureau au sous-sol)

Commande

Une commande est un programme binaire[1] qui effectue une tâche, souvent très simple et rapide : créer un dossier, supprimer ou déplacer des fichiers, etc.

Une commande est généralement issue d'un programme écrit en C et compilé grâce aussi une commande : **cc** (**C C**ompiler, Compilateur C) ou **gcc** (GNU C Compiler), que vous connaissez déjà pour en faire usage en TPs d'Algo/Prog.

Application

Une application est un programme (binaire) plus complexe que ne le sont les commandes. Par exemple, un éditeur de texte ou un navigateur Web sont des applications.

[1] Généralement c'est un binaire (le résultat d'une compilation). Dans de rares cas, il peut arriver que ce ne soit pas un binaire mais un script, c'est-à-dire un *fichier texte* contenant une succession d'autres commandes. Nous verrons les scripts plus tard. Pour le moment, on va considérer qu'une commande est un binaire. Ça ne change absolument rien dans les explications qui vont suivre.

Recette

Jusqu'à présent, on a vu comment on exécute des commandes (**`ls`**, **`mkdir`**, **`cd`** etc.), comment on ordonne au système de faire telle ou telle action, en tapant dans un Terminal le nom de la commande souhaitée, suivi éventuellement d'options et de paramètres.

Mais on ne s'est pas encore demandé quel mécanisme se produit quand on tape le nom d'une commande et qu'on appuie sur la touche **`ENTRÉE`**.

On l'a évoqué en début de TP, une commande est un programme binaire, résultat de la compilation d'un programme source. Vous avez déjà eu l'occasion, en TPs d'Algo/Prog, de produire des programmes binaires à partir de vos programmes sources, de cette façon :

```
cc exo1.c -o exo1 -Wall
```

Vous savez aussi que cette commande de compilation (**`cc`**) crée un fichier **`exo1`**, dans votre répertoire courant, et que pour exécuter votre programme, vous devez ensuite faire ceci :

```
./exo1
```

Cette étape, qui lance votre programme, est en tout point similaire au lancement d'une commande telle que **`cp`**, **`mv`**, **`rm`** etc., à l'exception près qu'on ne les fait pas précéder d'un **`./`** (on verra cela plus tard) mais sinon, c'est exactement la même chose : on exécute, on lance le programme (ici **`exo1`**) comme on lancerait une commande du système.

En fait, le programme (binaire) est stocké dans un fichier. C'est une sorte de *recette de cuisine informatique* qui contient la succession d'instructions que l'ordinateur doit réaliser pour produire le résultat attendu, par exemple : la création d'un dossier.

Comme toute recette (culinaire ou autre), ça reste une description d'étapes, d'actions à réaliser. Mais tant qu'on ne demande pas au chef de réaliser la recette, rien ne se passe, rien n'est produit.

Processus

Réaliser la recette, c'est produire un plat. Dans le cas de notre recette informatique, réaliser la recette c'est lancer la commande, c'est exécuter le programme.

L'ordinateur, par le biais du système, va lire la recette, c'est-à-dire lire le programme stocké dans le fichier de la commande à exécuter, et le placer en mémoire vive de l'ordinateur. A ce stade, le programme est prêt à s'exécuter, la recette va prendre vie. On passe d'un état de *recette* à un état de *plat*.

En informatique, *ce plat*, on appelle cela un *processus* :

- Un programme dans un fichier *sur disque* est une commande, une application, on dit aussi un *exécutable*.
- Un programme lu depuis un fichier et placé *en mémoire* pour être exécuté par l'ordinateur est un *processus*.

Où sont les commandes ?

Les commandes du système sont stockées à plusieurs endroits.

Il existe aussi une commande qui permet de savoir deux choses :

1. Si une commande existe, si elle est présente dans le système.
2. Où se trouve le fichier programme de cette commande, où est la recette.

La commande s'appelle **`which`** et s'utilise ainsi (exemple pour localiser **`ls`**) :

```
which ls
```

Q.1 Qu'affiche cette commande ?

Ça signifie que quand on demande au système d'exécuter la commande **`ls`**, il va lire la recette de la commande dans ce fichier qu'il trouve à ce chemin absolu précis.

Q.2 Maintenant, qu'affiche la commande suivante ?

```
which abcd
```

Quand rien ne s'affiche, ça signifie, en principe, que la commande n'existe pas dans le système, soit parce que vous avez fait une erreur dans le nom, soit parce que la commande n'est pas installée par exemple.

Testons maintenant avec la commande **`cd`** :

```
which cd
```

Rien ne s'affiche ! Ça signifierait donc qu'il n'y a pas de commande **`cd`** ? Ceci est plutôt bizarre car vous utilisez certainement cette commande **`cd`** depuis un moment, non ?!

C'est un cas spécial car **`cd`** n'est pas une commande du système mais une commande interne du Shell, c'est pour cette raison que rien ne s'affiche, il n'y a pas de *fichier recette* pour **`cd`**. Il y a quelques autres commandes de ce type, mais elles sont extrêmement rares, **`exit`** et **`logout`** sont deux autres exemples.

Les commandes du système sont dans :

- **`/bin`** et **`/usr/bin`** : signifient **bin**aries (binaires).
- **`/sbin`** et **`/usr/sbin`** : signifient **s**tatic **bin**aries (binaires statiques), en gros, les commandes réservées à l'administration du système, généralement utilisables par **`root`** seulement.
- **`/usr/local/bin`** : les commandes locales, des programmes spécifiques à votre organisation, à votre entreprise.
- **`~/bin`** : chaque utilisateur peut avoir son dossier **`bin`** personnel dans son *Home*. Il y place ses programmes personnels, à sa convenance.

Programmes personnels

Vous savez, depuis les TPs d'Algo/Prog, que vous devez faire précéder d'un **`./`** tout programme que vous souhaitez exécuter et qui est présent dans votre répertoire courant, car même si le Shell l'a devant son nez, il ne l'exécutera pas sans ce préfixe. C'est une chose étrange mais qui a son explication : par défaut (en l'absence du préfixe **`./`**) le Shell ne cherche à exécuter que les commandes qu'il trouve dans l'un des dossiers qu'on vient de lister ci-avant. Pour le forcer à exécuter un programme depuis le répertoire courant, il faut le dire explicitement en spécifiant un chemin relatif qui signifie *exécute tel programme qui se trouve ici* (**`./`**)

Affichage des processus

Un processus est donc un *truc vivant* dans la mémoire de l'ordinateur. On va pouvoir observer son comportement, ses actions, il va nous *dire* des choses (affichage), il va peut-être nous solliciter, nous poser des questions, et on va alors pouvoir interagir avec lui.

Si on prend le cas d'une application telle qu'un navigateur Web, *Firefox* par exemple, observer son comportement c'est voir les pages Web s'afficher, interagir avec lui c'est remplir un formulaire ou encore taper quelque chose dans la barre d'URL pour lui indiquer quel site, quelle page Web on souhaite consulter. On a bien un *truc vivant*, quelque chose qui est actif dans l'ordinateur. C'est ça un processus.

NE PAS CONFONDRE : Un processus n'est **pas un fichier**, comme une tarte n'est pas une recette. On utilise une recette pour produire une tarte. On utilise une commande, dont les actions sont décrites dans un fichier sur disque, **pour produire, en mémoire, un processus**.

Liste des processus

Pour lister les *fichiers sur disque*, on utilise la commande **`ls`**

Pour lister les *processus en cours d'exécution* en mémoire, on utilise la commande **ps** qui signifie **p**rocess(es) **s**tatus, état des processus.

Par défaut, la commande **ps** affiche la liste des processus de la session ouverte dans le *Terminal*. Pour faire simple, ce sont les processus qui ont été lancés[2] par ce Terminal qui est ouvert (sans doute dans une fenêtre), et uniquement celui-là. Testons :

```
ps
```

Q.3 Qu'affiche cette commande ? (Indiquez-le par écrit)

Chaque ligne représente un processus en cours d'exécution. Une ligne est composée (au moins) des champs suivants :

- **`PID`** : le **P**rocess **ID**, l'identifiant numérique unique de chaque processus.
- **`TTY`**[3] : le *Terminal* qui est attaché au processus. Pour faire simple, c'est le Terminal qui a lancé la commande qui a engendré ce processus. *Attaché* signifie que si le processus a besoin d'afficher des choses ou de lire au clavier, ce sera possible uniquement sur le **`TTY`** (le *Terminal*) attaché au processus.
- **`TIME`** : le temps d'exécution que le processus a consommé. On y revient un peu plus loin.
- **`CMD`** : la commande qui a été utilisée pour engendrer ce processus. On y trouvera aussi les options et les paramètres éventuels.

Pour l'instant, intéressons-nous uniquement au champ **`CMD`**, comprenez-vous la présence des différentes lignes qui viennent de s'afficher (avec **ps**) ? Faites-vous expliquer ou confirmer cela si besoin.

[2] Et sont toujours actifs, c'est-à-dire qu'il n'ont pas encore terminé leur exécution.
[3] **TTY** signifie **T**ele**TY**pe, le nom de l'équipement qui servait à taper des commandes à la préhistoire informatique, quand un Terminal ressemblait plus à une machine à écrire qu'à un écran de smartphone ! Pour info : la commande **`tty`** permet d'afficher le nom du Terminal (le bien nommé **TTY** justement) dans lequel on tape cette commande **`tty`**.

PID - Process ID

Le champ **PID** qui s'affiche avec la comme **ps** est le numéro unique de chaque processus.

Ce **PID** est unique à un instant T, mais il n'est pas associé à vie à une commande.

Quand on lance une commande, ça génère un processus en mémoire et ce processus se voit affecter un **PID** unique, le temps de son exécution. Une fois l'exécution terminée[4], le processus disparaît de la mémoire, le **PID** est à nouveau libre pour un futur processus.

Donc, si on relance la même commande après que la 1ère se soit terminée, la nouvelle commande aura un autre **PID**.

C'est un peu comme un ticket en file d'attente à la Sécurité Sociale, on retrouvera les mêmes numéros dès le lendemain, donnés à d'autres personnes.

Le nombre de **PID**s possibles[5] dans le système est limité, mais la limite est généralement suffisante pour les besoins communs.

TIME - Temps CPU

Le champ **TIME** qui s'affiche avec la comme **ps** est un temps exprimé en Heures, Minutes, Secondes : **HH:MM:SS**

Vous devriez être surpris de voir que le processus **bash**, par exemple, qui a été lancé en même temps que le *Terminal*, affiche un temps extrêmement faible, et en tout cas très éloigné du nombre de minutes réellement écoulées depuis que vous avez ouvert votre Terminal !

Ceci mérite donc une explication.

Le temps qui s'affiche est ce qu'on appelle du temps **CPU**[6]. On se rappelle que Linux est un *système multitâche* : il peut faire fonctionner, en parallèle, de nombreux programmes. Maintenant qu'on sait ce qu'est un processus, on va même pouvoir dire ceci : *Linux peut exécuter en parallèle de nombreux **processus***.

C'est le CPU qui a la charge d'exécuter les ordres successifs qui composent, qui font vivre, qui font s'animer les processus. Or, si de nombreux processus s'exécutent en même temps alors qu'il n'y a qu'un seul CPU, il va y avoir embouteillage dans le CPU, tout le monde ne va pas pouvoir être servi en même temps.

Linux, comme de nombreux autres systèmes, résout ce problème en *distribuant*, en allouant le temps par *minuscules tranches* (quelques millisecondes), successivement à

[4] Par exemple quand le programme a fait un **`return EXIT_SUCCESS;`** (cf TPs Algo/Prog).
[5] Souvent de l'ordre de plusieurs dizaines voire centaines de milliers.
[6] **CPU** signifie **C**entral **P**rocessing **U**nit, c'est l'appellation officielle du microprocesseur de l'ordinateur.

chaque processus. A l'échelle extrêmement rapide de l'électronique, ça donne l'impression que tous les processus s'exécutent en parallèle sans aucune pause. C'est un peu comme si on faisait avancer 10 brouettes l'une après l'autre de quelques centimètres en boucle, mais extrêmement rapidement, on aurait alors l'impression que les 10 brouettes avancent en même temps.

Pourtant, avec ce mécanisme de distribution du temps à tour de rôle, les processus passent beaucoup de temps (à l'échelle du CPU) à ne rien faire car ce n'est pas leur tour.

Il peut aussi arriver, et c'est même très fréquent, qu'un processus n'ait rien à faire à un instant T, parce qu'il attend, par exemple, que l'utilisateur tape au clavier. Le système exploite cela pour améliorer les performances du mécanisme de distribution du temps. Si des processus sont en attente de quelque chose (par exemple : quelque chose à lire au clavier, mais que l'utilisateur ne saisit rien pour le moment), alors ils n'entrent plus dans le partage du temps, jusqu'à ce que la chose qu'ils attendent soit arrivée (par exemple : que l'utilisateur ait finalement tapé au clavier).

Tout ceci fait qu'en cumulé un processus ne travaille que très peu de temps (comme un étudiant finalement 😆) et que la majeure partie du temps réel, le temps qui s'écoule pour un être humain, il… dort !

Le champ **`TIME`** n'affiche pas le temps réellement écoulé à notre horloge d'humain, mais le temps cumulé qui a été consommé par le processus pour s'exécuter dans le CPU (la somme des petites *tranches de temps*).

C'est la raison de ces énormes différences de temps affichées par la commande **`ps`**.

Tous les processus

La commande **`ps`**, sans option, affiche la liste des processus lancés depuis un Terminal donné, mais on sait qu'il y a beaucoup d'autres processus qui s'exécutent en même temps, en parallèle, dans le système. Par exemple, vous lisez sans doute ce PDF dans une liseuse de PDF, c'est une application, elle est donc en train de s'exécuter sur votre ordinateur, et pourtant le processus de la liseuse n'apparaît pas dans la liste que vous avez sous les yeux.

Pour afficher tous les processus du système, on peut passer des options à la commande **`ps`**. Cette commande dispose de beaucoup d'options (**`man ps`**). Chacun a ses habitudes, ses options préférées, voici une suggestion pour afficher tous les processus du système, il existe d'autres combinaisons d'options qui font, peu ou prou, la même chose. Celle-ci à l'avantage, pour nous français, d'être facile à mémoriser :

```
ps -edf
```

Évidemment, rien à voir avec une quelconque entreprise du domaine de l'énergie électrique. Rappel : **`-edf`** est un raccourci pour **`-e -d -f`**. Sans y passer trop de temps, consultez le manuel pour plus de détails sur chacune de ces 3 options.

L'option intéressante est ici **`-e`**, qui affiche tous les processus du système.

Nous verrons dans quelques semaines comment sont gérés les droits sur les objets dans Linux. Par exemple, on peut avoir le droit de lire certains fichiers et pas d'autres. Mais, en ce qui concerne les processus, tout le monde peut consulter la liste des tous les processus en cours d'exécution sur le système. C'est ce que fait le **`ps -edf`** ci-dessus.

Dans cette liste, il y a plus d'informations qui s'affichent pour chaque processus, c'est le **`-f`** qui nous donne ça. Parmi ces informations, on retrouve déjà celles qu'on a détaillées précédemment. On va s'intéresser maintenant à certaines autres :

- **`UID`** est le propriétaire du processus, l'utilisateur qui a lancé la commande qui a engendré le processus.

Q.4 Essayez déjà d'identifier quelques-uns des vôtres.

- **`PPID`** (attention à ne pas le confondre avec **`PID`**) est le **`PID`** du processus parent. On détaille ci-dessous.

PPID - Parent PID

Les processus sont comme le *système de fichiers*, ils forment une **arborescence**. Un processus a toujours un parent. Cela signifie qu'un processus est toujours engendré par un autre processus, qui lui-même a été engendré par un autre processus et ainsi de suite. C'est le mode de fonctionnement de Linux. Un processus ne naît pas de nulle part.

Quand vous lancez une commande :

```
ps
```

vous avez certainement remarqué que dans la liste des processus qui s'affiche apparaît aussi ce processus **`ps`**. C'est normal, comme la commande **`ps`** engendre un processus qui va lister les processus en cours, ce processus se voit lui-même en cours d'exécution. C'est un peu comme faire un selfie.

Q.5 Une question se pose maintenant : on vient de dire que tout processus est engendré par un autre processus qu'on appelle son *parent*. Quel est donc le processus parent du processus **`ps`** qui s'affiche dans la liste ? Avez-vous une idée (uniquement à partir du résultat de **`ps`** sans option) ?

Pour vérifier votre intuition, affichez la liste de tous les processus et repérez votre processus **`ps -edf`**, puis repérez le **`PPID`** (**`PID`** du parent) sur la ligne et recherchez ce **`PPID`** dans les **`PID`**s des autres processus pour trouver le parent. Alors, votre intuition

était-elle la bonne ? Si vous ne trouvez pas ou ne comprenez pas, demandez à votre enseignant.e de vous aider.

Q.6 Ouvrez une application **Geany**[7] depuis l'interface graphique. Dans un Terminal, affichez tous les processus[8]. Remontez un par un les processus ascendants de votre **Geany**. Reconstruisez sur papier les noms des processus qui forment l'arbre généalogique de ce processus **Geany**. Où aboutissez-vous ? Essayez avec un autre processus (par exemple vous pouvez lancer un Visual Studio Code, le processus s'appellera **code** en principe). Quelle est votre conclusion ?

Ce processus ancêtre, que vous devriez avoir découvert dans l'exercice précédent, s'appelle généralement **systemd** ou **init** sur certains systèmes. Il est essentiel au bon fonctionnement du système, c'est lui qui amorce le système tout entier. S'il venait à s'arrêter, ça ne serait pas très bon pour la santé du système, il faudrait certainement redémarrer l'ordinateur, mais ça arrive extrêmement rarement, heureusement !

Affichage de l'arborescence de processus

La commande **ps** possède des options pour afficher cette arborescence de façon plus visuelle :

```
ps -axuwf
```

Encore une fois, il existe d'autres combinaisons d'options qui arrivent à quelque chose de similaire. Chacun a ses combinaisons d'options préférées. Celle ci-dessus est une simple proposition, qui fonctionne plutôt bien.

Voici une autre commande permet de faire la même chose :

```
pstree -pshl
```

Q.7 Repérez votre morceau d'arborescence depuis le processus **Geany** de l'exercice précédent.

Avant de terminer sur ce sujet de l'arborescence des processus, repérez quand même que le tout 1er processus (**systemd** ou **init**) n'a pas de parent ou plutôt (**ps -edf**) que

[7] **Geany** est un éditeur de texte mais on ne va pas y saisir quoi que ce soit, c'est juste histoire d'avoir une application graphique à lancer pour l'exercice.
[8] Vous devez relancer un affichage de tous les processus une fois **Geany** lancé, même si vous aviez déjà une liste de processus affichée dans votre Terminal, car la liste évolue en permanence au gré des lancements et des arrêts des processus. Ce n'est pas une liste figée comme pourrait l'être une arborescence de fichiers, ça bouge très fréquemment et très vite.

son **PPID** est zéro et que le **PID** zéro ne correspond à aucun autre processus. Ce zéro signifie que c'est le noyau, le kernel (cf le CM 1), qui a engendré ce processus. Le kernel n'est pas un processus.

Actions sur les processus

On a vu que n'importe quel utilisateur peut lister tous les processus actifs.

Par contre, seul celui qui a lancé un processus a le droit de faire des actions sur ce processus. NB : **root** a tous les droits.

Mais que signifie *faire des actions* ?

Généralement il s'agit de mettre un terme au processus. On appelle cela : *tuer le processus*. Heureusement donc que seul le propriétaire a le droit de faire cela, sinon ça poserait de gros problèmes de sécurité, surtout sur un système multi utilisateurs comme l'est Linux !

Q.8 Pour pouvoir faire une action sur un processus, il faut avant tout identifier la cible. Qu'est-ce qui identifie de manière unique un processus ? Faites-vous confirmer cela par l'enseignant.e car tout ce qui suit en dépend.

Quand on fait une action sur un processus, on lui envoie un signal. Il existe plusieurs signaux et chacun a sa fonction propre. On verra en détail les signaux en BUT2. Cette année on se limitera à 3 d'entre eux.

Préparation

Pour la suite de ce TP, nous allons avoir besoin d'exécuter un petit programme que vous allez devoir compiler. Voici le code source (**on ne vous demande PAS de comprendre ce qu'il fait**). Il est récupérable sur Moodle.

```
#include <stdio.h>
#include <unistd.h>

const int BEAUCOUP = 999999;

int main() {
     int loop;

     for (loop = 1; loop < BEAUCOUP; loop++) {
          sleep(1);
          printf("Ca fait %d seconde(s) que je tourne...\n", loop);
```

```
        }
}
```

Compilez-le pour produire un exécutable **tictac**, puis exécutez-le :

```
./tictac
```

Vous devez observer un affichage chaque seconde, sans fin ou presque[9].

kill

La commande qui permet d'envoyer un signal à un processus s'appelle… **kill**

Cette commande ne porte pas un bon nom car elle permet d'*envoyer un signal*, et non pas de tuer un processus ! Alors, pourquoi porte-t-elle ce nom ambigu ? En fait, la plupart des signaux ont pour effet, par défaut, de mettre un terme au processus, de le tuer. Ceci explique donc le nom de la commande. On verra en BUT2 que cet effet peut être détourné si la commande le souhaite.

Les signaux portent des noms et des numéros. Voici la syntaxe :

```
kill -<SIGNAL> <PID>
```

où **<SIGNAL>** est soit le *numéro* soit le *nom* du signal à envoyer et **<PID>** est le **PID** du processus ciblé (au passage, vous avez ici la réponse à la Q.8). Notez bien le **-** devant le numéro ou nom du signal.

On ne va pas lister ici tous les signaux possibles car ce sera étudié en détail en BUT2. On va juste s'intéresser au signal **9**, dont le nom est **KILL** (oui, cette fois-ci le nom correspond parfaitement à l'action qu'il fait).

> Sauvez bien vos fichiers en cours si besoin, notamment vos notes si vous en prenez dans VSC car si vous ratez votre coup, vous pourriez tuer le mauvais processus et perdre ce qui n'aura pas été sauvé ! C'est cool le système, on peut faire plein de bêtises rigolotes…

Nous allons tester cette commande **kill** sur le processus **tictac** qui doit être actuellement en train de s'exécuter (si ce n'est pas le cas, lancez-le dans votre Terminal).

[9] Si on le laisse, il va tourner 999.999 secondes, soit environ 11,5 jours

Cette fois-ci on est un peu bloqué car le Terminal est occupé par notre programme **tictac** qui tourne sans fin. Impossible de saisir une autre commande avant que le processus en cours (**tictac**) ne soit terminé.

Qu'à cela ne tienne, ouvrez un second Terminal, et dans ce nouveau Terminal, identifiez (cherchez dans le résultat d'un **ps**) le **PID** du processus **tictac** et tuez-le avec la commande **kill**. Vous avez donc 2 solutions (remplacez **<PID>** par la bonne valeur !) :

```
kill -9 <PID>
```

ou bien :

```
kill -KILL <PID>
```

Vérifiez dans le 1er Terminal que le processus **tictac** s'est bien interrompu.

Pour les plus curieux, la commande :

```
kill -L
```

affiche tous les signaux qui existent. Pas de panique, ça ne vous sera d'aucune utilité en BUT1, c'est juste pour vous donner un avant goût de l'avenir !

Au clavier

Il existe une autre façon de mettre un terme à un processus, sans passer par la commande **kill**, directement par une action au clavier. Il suffit d'appuyer simultanément sur les touches **CTRL** et **C**. On appelle cette action *faire un CONTROL-C*, et on l'écrit (dans un sujet de TP par exemple) : **CTRL-C**.

Cette action doit se faire dans le *Terminal* qui exécute actuellement le processus à interrompre, à tuer.

Relancez **tictac** et essayez, dans le même Terminal, l'action d'un **CTRL-C** sur lui.

Avez-vous réussi à arrêter le processus ? Bravo. Dans le cas contraire, demandez à votre enseignant.e de vous y aider.

Note : un **CTRL+C** dans le Shell ne tue pas le Shell. D'ailleurs certains programmes (généralement pas les commandes du système) peuvent refuser de *mourir* de cette façon. On verra en BUT2 comment procéder.

Pour information, un **CTRL+C** envoie un signal **15** qui s'appelle **TERM** (comme **TERM**inate) au processus. Ceci constitue le second signal qu'on verra cette année. Cela signifie qu'on pourrait aussi envoyer ce signal à l'aide d'un **kill -TERM <PID>**

Asynchronisme

Il nous reste une dernière chose à voir concernant les processus : l'asynchronisme.

L'asynchronisme est le caractère de ce qui ne se déroule pas à la même vitesse ou, plus précisément, de choses qui se déroulent à des vitesses indépendantes les unes des autres.

Linux est un système globalement asynchrone. Chaque processus se déroule à son rythme, indépendamment des autres processus. Un processus donné ne dépend généralement pas d'un autre processus, sauf si on a volontairement codé ces processus pour qu'ils soient synchronisables, une particularité qui sera étudiée en BUT2.

L'asynchronisme se matérialise, par exemple, par le fait que télécharger un fichier depuis un site Web dans son navigateur n'impactera pas l'écoute simultanée d'un morceau de musique, ni la compilation d'un programme qui s'effectue parallèlement dans un autre Terminal. Toutes ces choses sont asynchrones. Aucune ne dépend des autres pour se dérouler.

Shell

Cependant, quand on est dans le Shell (dans un Terminal), si on lance une commande on ne peut plus en lancer une seconde tant que la 1ère n'est pas terminée. Par défaut, le Shell fonctionne en mode synchrone et non pas asynchrone. Si on veut lancer deux commandes simultanément, on doit ouvrir deux Terminaux.

Pourtant, il existe une solution, en plaçant un **&** en fin de ligne après la commande avant de taper sur la touche **ENTRÉE**, ainsi :

```
commande &
```

Évidemment, ceci n'a d'intérêt que si la commande met un peu de temps à s'exécuter, sinon le temps de taper la commande suivante, la précédente aura déjà terminé. On ne peut donc pas tester ça sur un **ls**, un **rm** ou un **mkdir**.

Notre **tictac** est parfait pour l'expérience.

Dans un premier temps, lancez ceci (on n'utilise pas de **&**) :

```
./tictac
```

Le processus commence à afficher ses messages chaque seconde. Tentez de lancer ceci ensuite (attention, vos saisies au clavier peuvent se télescoper avec les affichages de **`tictac`** mais ce n'est pas grave, ne cherchez pas à corriger quoi que ce soit, les frappes au clavier sont bien prises en compte indépendamment et même si elle se mélangent avec les affichages de **`tictac`**) :

```
ls -l
```

N'oubliez pas de valider par la touche **ENTRÉE**.

Q.9 Obtenez-vous un **`ls`** du dossier courant ?

Tuez le processus avec un **CTRL+C**

Maintenant, lancez ceci (on utilise un **&** cette fois-ci) :

```
./tictac &
```

Le processus commence encore à afficher ses messages chaque seconde.

Lancez maintenant ceci :

```
ls -l
```

Q.10 Obtient-on un **`ls`** du dossier courant cette fois-ci ? (Il est possible que les 2 affichages s'entremêlent)

Effectivement, on observe que **`tictac`** ne bloque plus le Terminal. La commande **`tictac`** s'exécute de façon asynchrone, indépendamment de la commande **`ls`**. Aucune des deux n'impacte l'autre.

Attention : les affichages s'entremêlent mais ça ne veut pas dire qu'ils s'impactent l'un l'autre, c'est juste que ces deux commandes utilisent le même moyen d'afficher leurs informations, l'écran, le Terminal. Un peu comme si deux étudiants écrivaient sur le même tableau blanc en même temps, les écritures arrivent au même endroit mais les deux étudiants travaillent simulatanément et indépendamment l'un de l'autre.

Dans le Shell, on appelle cela : *lancer une commande en arrière-plan* (background).

Q.11 Tuez maintenant le processus tictac avec un **CTRL+C**. Que se passe-t-il ?

Pour pouvoir arrêter un processus avec un **CTRL+C**, il doit être en *avant-plan* et pas en *arrière-plan*. Or, on a lancé un **tictac &** et on vient de voir que ce **&** permet d'exécuter le processus en le plaçant en *arrière-plan*.

Deux solutions s'offrent à nous pour y remédier :

- Tuer le processus avec un **kill**
- Remettre le processus **tictac** en *avant-plan* pour qu'il puisse recevoir le **CTRL+C**

Arrière et avant-plans

On sait déjà comment faire un **kill**, on va donc plutôt s'intéresser à la façon dont on peut remettre un processus en *avant-plan* dans un Terminal.

Il suffit de taper la commande suivante dans le Terminal dans lequel se trouve un processus actuellement en *arrière-plan* :

```
fg
```

fg signifie **f**ore**g**round (avant-plan). Le processus d'*arrière-plan* va alors continuer son exécution comme si on n'avait pas mis de **&** à la fin lors de son lancement. Il est repassé en *avant-plan*, il devient de nouveau réactif au **CTRL+C**. Testez-le.

L'inverse est aussi possible. Lancez maintenant ceci :

```
./tictac
```

Le processus **tictac** s'exécute en *avant-plan* (pas de **&** à la fin).

Il est maintenant possible d'envoyer ce processus en *arrière-plan* comme si on avait placé un **&** en fin de ligne dès le lancement. Voici comment on procède :

- Il faut d'abord taper un **CTRL+Z** (même technique qu'avec **CTRL+C**). Cette action au clavier met le processus *en pause*, c'est-à-dire que le système ne lui donne plus de temps CPU pour s'exécuter (la brouette reste sur place). Le processus n'est pas tué, il est juste en sommeil, mais placé en *arrière-plan*.
- Ensuite, il suffit de taper la commande **bg** (**b**ack**g**round) qui va le *réveiller* mais en le conservant en *arrière-plan*, rendant du coup la main à l'utilisateur dans le Terminal, comme avec l'usage d'un **&**.

Testez-le.

Notez que **fg** et **bg** ne sont pas des commandes du système mais des commandes du Shell. La notion d'*arrière-plan* et d'*avant-plan* n'ayant de sens que dans le cadre du Shell.

Pour information, un **CTRL+Z** envoie un signal **20** qui s'appelle **TSTP** (comme **T**erminal **ST**o**P**) au processus. Ceci constitue le troisième et dernier signal qu'on verra cette année.

Quels usages ?

Lancer en *arrière-plan* un processus qui affiche des choses à l'écran a assez peu d'intérêt car ses affichages vont venir polluer la saisie de commandes et l'affichage de ces autres commandes dans le Terminal. Il vaut mieux privilégier l'utilisation de plusieurs Terminaux dans ce cas.

Il existe des cas plus intéressants, soit quand la commande lancée en *arrière-plan* ne produit pas d'affichage, soit quand le processus lancé dispose, par exemple, d'une interface graphique.

Expérimentez avec VSC en lançant ceci :

```
code &
```

Ça devrait vous lancer une nouvelle fenêtre VSC et vous permettre de conserver la main dans votre Terminal pour continuer à saisir des commandes.

Nous aurons aussi l'occasion, dans un prochain TP, de voir comment on peut empêcher les affichages produits par les commandes de venir se télescoper.

Il ne peut y avoir qu'un seul processus en *avant-plan* mais il peut y avoir plusieurs processus en *arrière-plan*.

Note importante : tout processus d'*arrière-plan* qui tente de lire au clavier est automatiquement et instantanément placé *en pause* par le système (comme si on avait tapé **CTRL+Z**).

Le clavier est toujours associé à un processus d'*avant-plan*.

En l'absence de processus d'*avant-plan*, c'est le Shell qui dispose du clavier. C'est pour cela qu'un **CTRL+C** ou un **CTRL+Z** ne fonctionne que sur un processus d'*avant-plan*.

Les programmes utilisant une interface graphique font exception car ils ont accès au clavier d'une façon différente et ne sont pas concernés par cette remarque.