Menu du jour : les processus

- ce qu'ils sont
- comment ils sont gérés par le SE
- ce qu'il se passe à leur naissance, pendent leur vie, et à leur mort.
- comment on peut les faire communiquer (signaux et pipes)

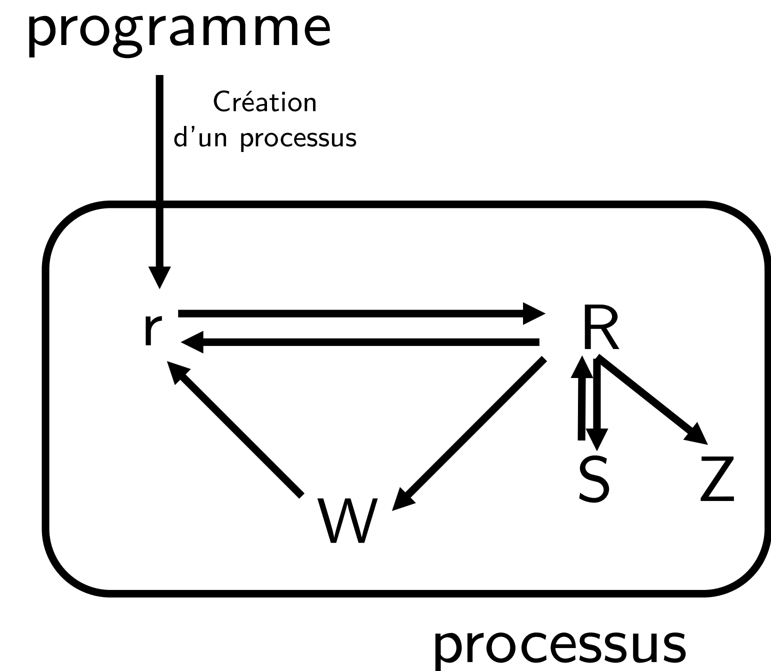
Les processus : définitions

Introduction aux processus

- Un **programme** est un fichier contenant du code pouvant être exécuté. Exemple : le **a.out**
- Un **processus** (process) = une **instance** d'un programme en cours d'exécution
- Pour être exécuté, un programme est chargé dans la mémoire vive, ses instructions sont exécutées par le processeur. Le système d'exploitation lui fourni un **espace d'addressage** (une zone de mémoire fournie pour qu'il puisse puisse écrire dedans).
- Un système d'exploitation **multitâche** doit traiter plusieurs processus en même temps.
- **ps -aux [| grep *nomprocessus*]**

L'ordonnanceur et les états possibles

- L'**ordonnanceur** (**scheduler**) = partie du noyau qui choisit quel processus doit s'exécuter à un moment donné
- Il maintient :
  - **Le** processus en train de s'exécuter (**R** – Running)
  - **Une file** de processus prêts à s'exécuter (**r** – runnable)
  - **Un ensemble** de processus en attente d'un événement (**W** – waiting)
  - **Un ensemble** de processus endormis (**S** – sleeping)
  - **Un ensemble** de processus morts (**Z** – zombie)
- Les infos sur les processus sont gérés dans la **table de processus**



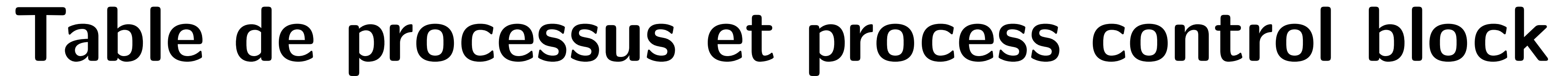


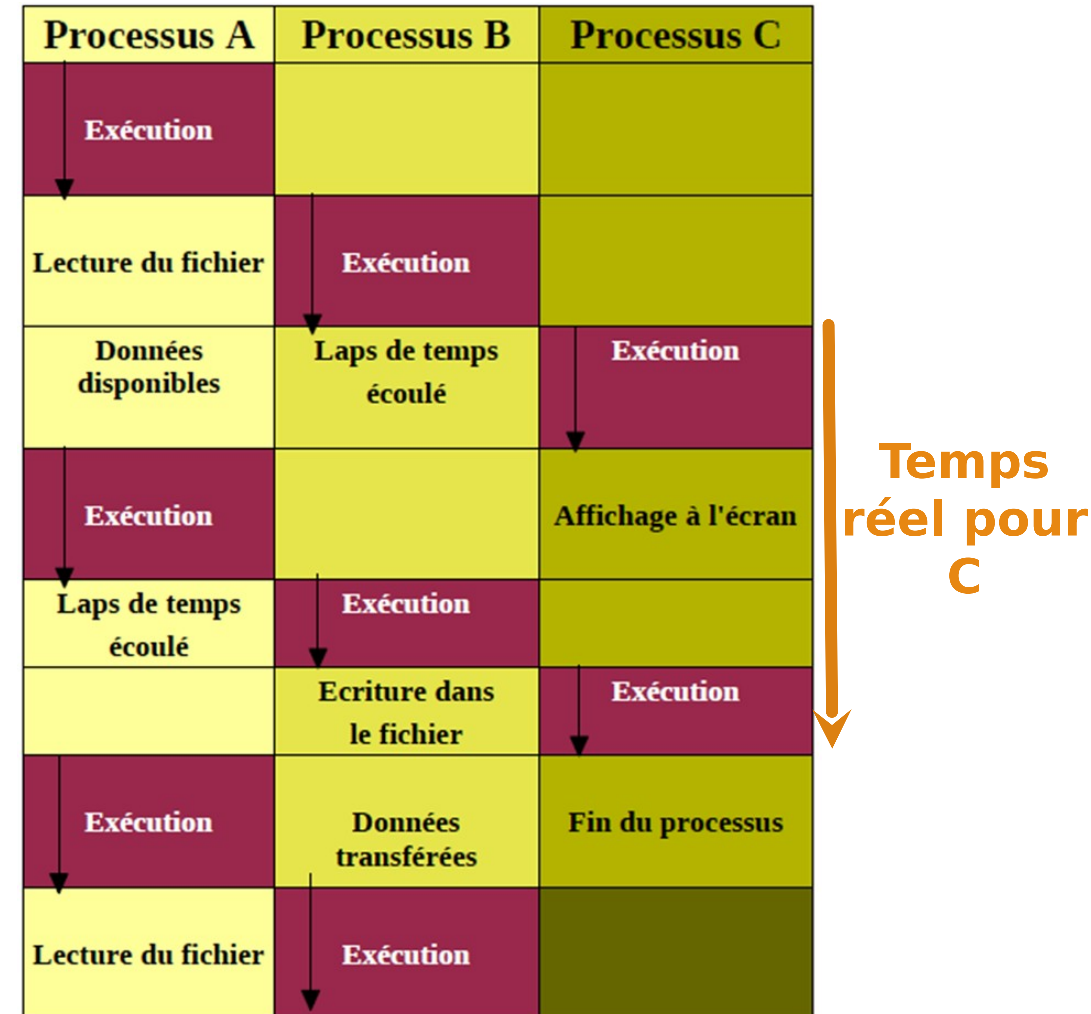
Table de processus et process control block

Table de processus = un tableau où chaque case (appelée un process control block) conserve, pour un processus donnée, les infos qui doivent toujours être accessibles par le noyau

- Identité du processus : **PID** : process identifier, **PPID** : parent PID
- **Propriétaires** : **réels** (qui on est vraiment) et **effectif** (de qui on a les droits) (cf Exo 1 TD 4)
- **État du processus** (cf diagramme d'avant)
- Le répertoire courant = une référence à un i-node
- **Avancement** du processus = l'état de la mémoire utilisée, la valeur de ses variables, l'adresse de la prochaine instructions, la liste des fichiers ouverts ...
- **Priorité**,
- Autre : événements attendus par le processus, vecteur de signaux que le processus n'a pas encore géré, ...

Changement de processus

- UNIX est un **système multitâche à temps partagé** (cours 2). Comme il n'a (en général) qu'un seul processeur, il ne peut traiter qu'une tâche à la fois. Pour donner l'illusion du parallélisme, il **commute** rapidement entre les tâches.
- Sur un intervalle de temps assez grand, tous les processus ont progressé, mais **à un instant donné un seul processus est actif.**
- **Temps réel** écoulé != **temps CPU**



Commutation de contexte

Chaque commutation entre deux processus P1 et P2 nécessite de sauvegarder l'état d'avancement de P1, et de reprendre où P2 s'était arrêté

Si pour une raison x ou y, le système d'exploitation a décidé de changer quel processus est exécuté, le noyau :

- Arrête l'exécution du processus de x,
- Copie les valeurs des registres hardware dans le process control block (sauvegarde du **mot d'état**)
- Charge le mot d'état du processus y (màj de registres avec les valeurs du processus y)
- Lance l'exécution de y

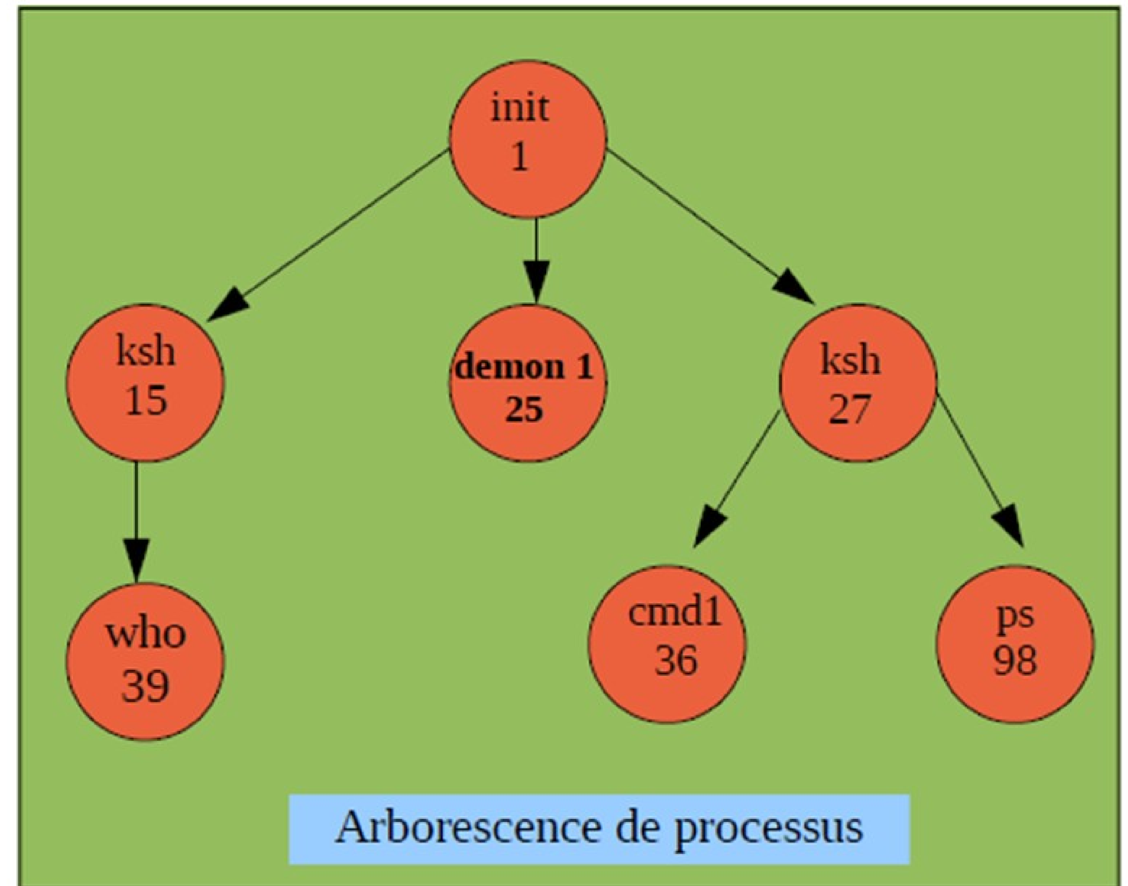
Temps nécessaire pour changer de contexte : **overhead**

init, le processus originel

- PID $= 1$
- Lancé par le kernel pendant la phase de boot du système (**systemd** / sysVinit / upstart / ...)
- Lance les scripts de démarrage (fichier rc dans `/etc`), monte le système de fichiers, démarre tous les services nécessaires (démons), crée un processus par terminal (tty) qui attend une connexion de l'utilisateur.ice.

L'arborescence des processus

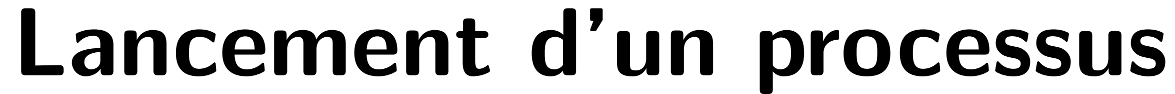
- Si une connexion réussi, le processus de login exécute un shell qui peut accepter des commandes.
- Ces commandes peuvent lancer d'autres processus, ...
- Donc : **les processus lancent des processus qui lancent des processus qui** ... $\rightarrow$ un **arbre** (/!\ ne pas confondre avec l'arborescence des fichiers)
- Tous les processus dérivent de init.



Les processus
Comment ils naissent, vivent, et meurent

Cycle de vie des processus

- Sauf pour init : un processus parent génère un nouveau processus (mécanisme de **fork** ou de **clone**)
- Le processus fait sa vie
- Le processus meurt
  - Soit par sa propre volonté : fin d'exécution normale (retourne 0), ou erreur non critique (retourne une valeur $!= 0$)
  - Soit involontairement : via une erreur critique, ou une interruption provoquée par un autre processus



Lancement d'un processus

- Dessin : un processus fait un fork
- Il y a donc 2 processus :
  - Le **processus père P**, qui exécute le programme Shell,
  - Le **processus fils F**, qui exécute la commande.
- Le fils **hérite de tout l'environnement** du processus file père, <u>**sauf**</u> du **PID,** du **PPID** et des **temps d'exécution**.
- Cas 1 : F finit avant P, le SE notifie P qui est tâché de gérer
- Cas 1' : P était en wait mode.
- Cas 2 : P finit avant F, le SE rattache F à init

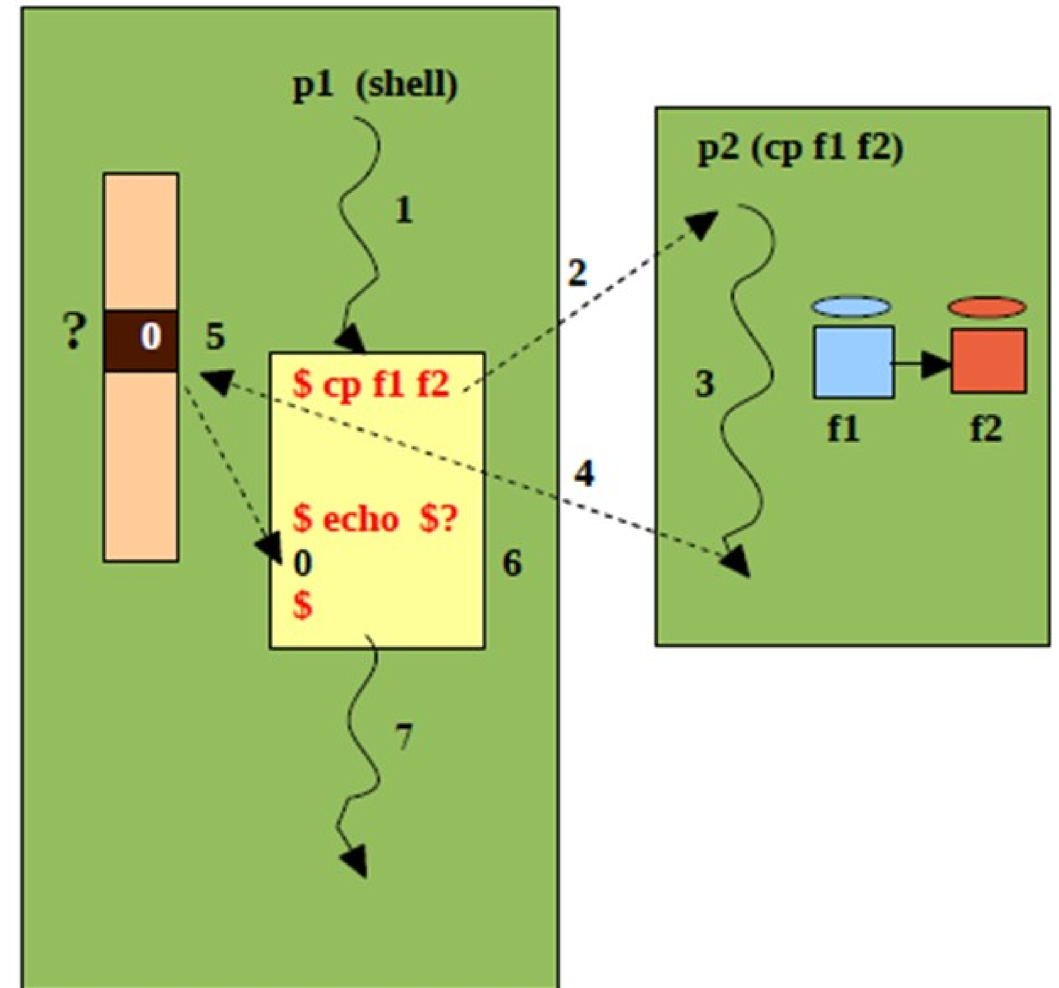
Cycle de vie des processus : illustration dans le shell

Pour chaque commande lancée*, le Shell crée automatiquement un nouveau processus et se met en attente.

Exemple :

prompt> **cp f1 f2**
prompt> **echo $?**
0

* : sauf les primitives, qui sont directement intégrées au shell, voir TD...



Commande shell en premier plan

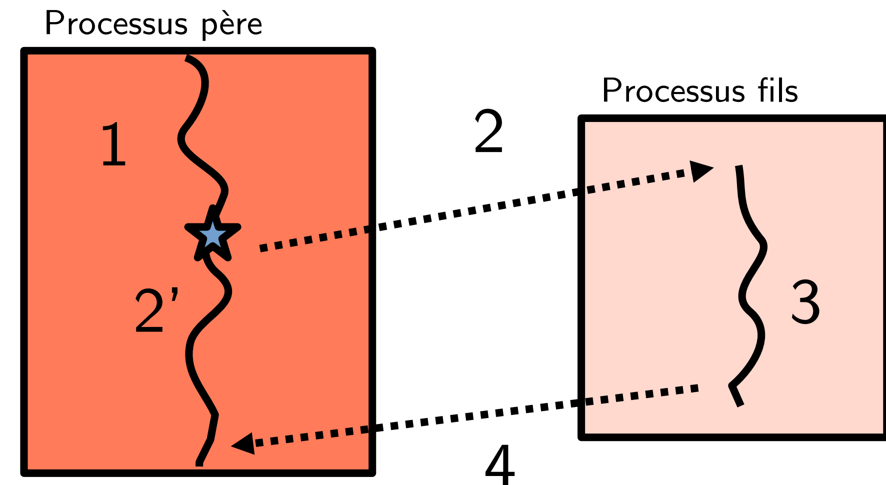
- C'est le mode par défaut : on entre une commande, on attend la réponse : **le shell nous redonne la main quand le processus est fini**.
- Tant que le processus fils qui correspond à la commande n'est pas fini, le processus père (le shell) est en attente → on ne peut pas utiliser le shell.

prompt> **cmd1**
… résultat de la commande cmd1
ça peut être long
trèèèèèèès lonnnnng.
trop long ? (faites Ctrl-C si vous en avez marre)

Processus père
Processus fils
1
2
3
4
5

Commande shell en arrière plan

On utilisera cette solution (processus lancés en parallèle) par exemple pour lancer un traitement très long, et continuer à travailler en même temps. Dans ce cas, on dit que le père a lancé un fils en **tâche de fond** (**background**) ou encore en **mode asynchrone**.



Commande shell en arrière plan

prompt> **cmd1 &** # le nom de la commande suivi de '&'
[1] 127
prompt >

Le Shell affiche un **numéro de tâche** entre « [ ] » et le **PID** de cette tâche de fond, puis continue à travailler (→ donc affichage de la chaîne d'invite et attente de la prochaine commande).

Pour lancer plusieurs commandes successives (« ; ») en arrière plan :

prompt> **(cmd1; cmd2) &**
[2] 128
prompt>
→ La commande **cmd2** ne sera lancée que lorsque la commande **cmd1** sera terminée. L'utilisateur.ice récupère la main tout de suite. Le Shell détecte la présence du **'&'** partout sur la ligne.

Commande shell en arrière plan

- Dans le cas suivant, la commande **cmd1** est lancée en arrière plan et la commande **cmd2** est tout de suite lancée derrière, en direct (en parallèle).

```
prompt> cmd1 & cmd2
 [3] 130
 résultat commande 2
prompt>
```

- La commande « **wait *n*** » permet d'attendre la mort de la tâche de fond dont le PID est « ***n*** ». .

```
prompt> cmd1 &
 [4] 132
prompt> wait 132 # rester bloqué jusqu'à ce que cmd1 se termine
```

  Si « ***n*** » n'est pas précisée, **wait** attend la mort de toutes les tâches de fond. **wait** ne s'applique qu'aux processus lancés dans le shell lui-même.

- Pour lister les processus lancé dans la session en cours : **jobs**

La communication entre les processus

Communications inter-processus

Deux paradigmes :

- par structures de données partagées.

  Exemple : via des fichiers en lecture-écriture concurrentes, via des bases de données, ...
- par messages

  Exemple : via les signaux et les tubes

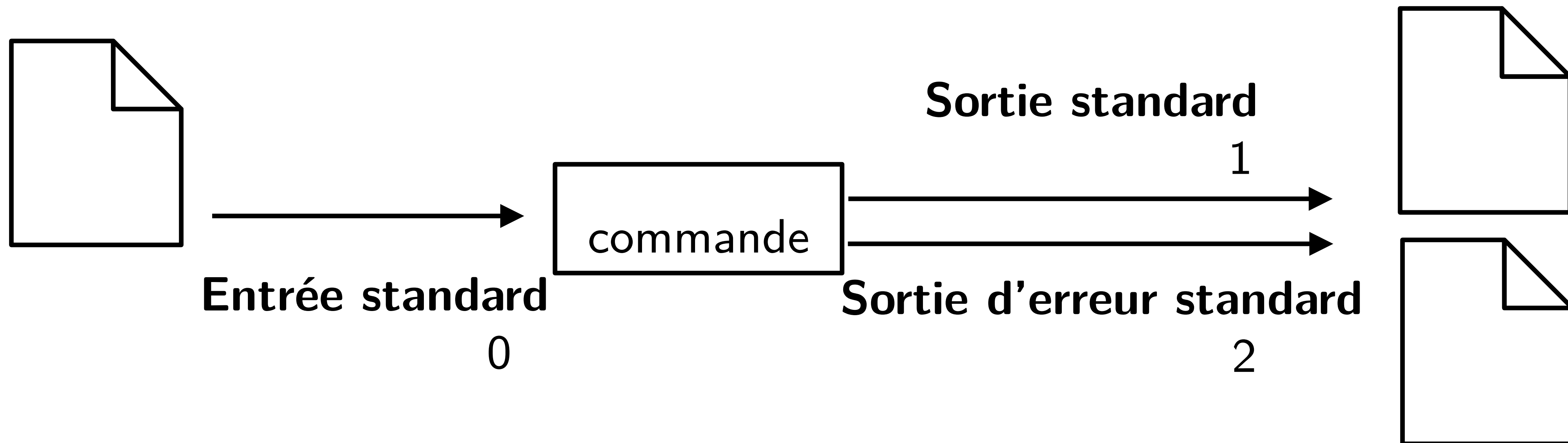
Communication via signaux

- Les processus communiquent (entre eux et avec le SE) via des signaux
- Liste de signaux : man7.org/linux/man-pages/man7/signal.7.html

  exit et **return** : « coucou kernel, j'ai fini » $\rightarrow$ fin d'exécution normale
  SIGTERM : sommation d'interruption
  SIGINT : interrompu en douceur (généralement CTRL-C dans le terminal)
  SIGKILL : interruption violente
- Dans un processus, un **handler** c'est une fonction qui attrape les signaux et les gère.
  Exception : **SIGKILL** et **SIGSTOP**
  le CTRL-C dans vim ne tue pas le processus

Communication via pipe

Rappel sur les entrée / sortie des commandes



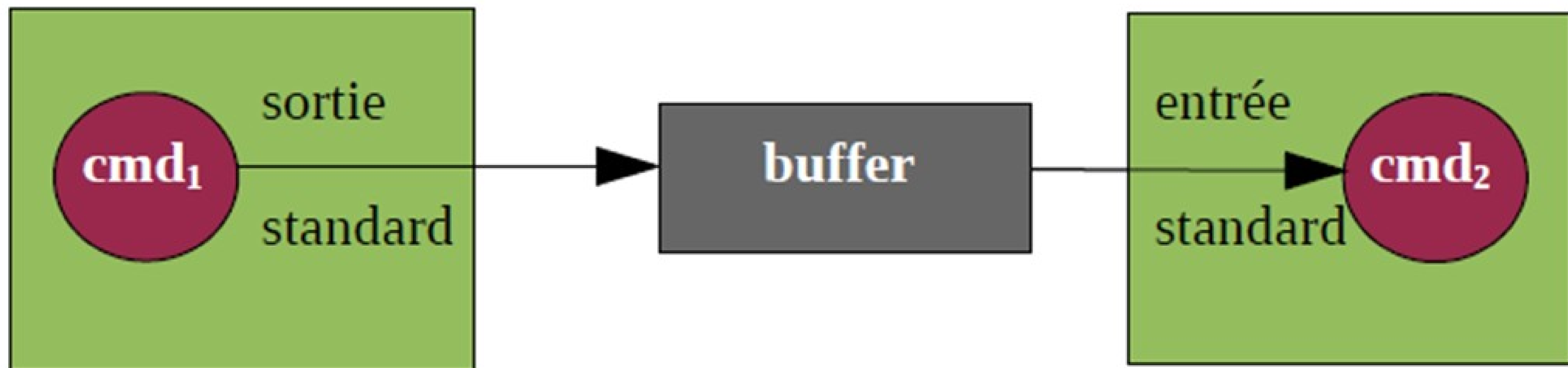
Communication via pipe

Un pipe (tube) permet de faire communiquer deux processus en pluggant la sortie de A sur l'entrée de B.

Les deux processus s'exécutent en parallèle.

En shell : **commande_1 | commande₂ [| ... | commande_n]**

Exemple : **ls | grep toto.txt**



Communication entre processus via un pipe

prompt> **who | grep cours**

cours	ttya4	Jul 31 10:50
cours	ttyc6	Jul 31 09:34
cours	ttya2	Jul 31 09:02

→ La sortie produite par la commande **who** est associée à l'entrée de la commande **grep**. **who** donne la liste des personnes connectées au système à un moment donné ; **grep** cherche si la chaîne cours est présente dans le flot de données qu'elle reçoit. On peut donc considérer que la commande **grep** joue le rôle de filtre.

Le pipe est plus court et compact que :

prompt> **who > tmp**

prompt> **grep cours < tmp**

cours	ttya4	Jul 31 10:50
cours	ttyc6	Jul 31 09:34
cours	ttya2	Jul 31 09:02

prompt> **rm tmp** # pour ne pas conserver le fichier intermédiaire

Communication entre processus via un pipe

prompt> **ps -a | wc -l**
9

Création de deux processus concurrents. Un tube est créé dans lequel le premier (**ps -a**) écrits ses résultat et le deuxième (**wc -l**) lit.

Lorsque le processus écrivain se termine et que le processus lecteur dans le tube a fini d'y lire (le tube est donc vide et sans lecteur), ce processus détecte une fin de fichier sur son entrée standard et se termine.

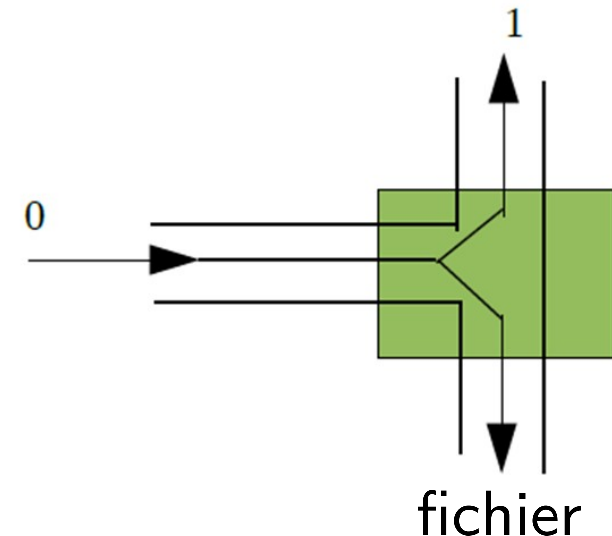
Le système assure la **synchronisation de l'ensemble** dans le sens où :

- il bloque le **processus lecteur** du tube lorsque le tube est vide en attendant qu'il se remplisse (s'il y a encore des processus écrivains);
- il bloque (éventuellement) le **processus écrivain** lorsque le tube est plein (si le lecteur est plus lent que l'écrivain et que le volume des résultats à écrire dans le tube est important).

Communication entre processus via un pipe
La commande tee

- **cmd1 | cmd2 | cmd3 | .... | cmdn**
- On ne voit pas les résultats intermédiaires
- **tee** : lit dans son entrée standard (0), écrit dans sortie standard (1) et dans un fichier
- Exemple : **ps -l | tee /dev/tty | wc -l**

  F S UID PID PPID PRI ... CMD
  1 S 102 241 234 158 -bash
  1 R 102 294 241 179 ps
  1 S 102 295 241 154 tee
  1 S 102 296 241 154 wc
  5



Communication entre processus via un pipe
Les filtres

Un **filtre** est une commandes ayant la propriété **à la fois de :**

- **lire sur leur entrée standard et**
- **d'écrire sur leur sortie standard**.

Commandes filtres : **cat**, **wc**, **sort**, **grep**, **sed**, **sh**, **awk**, **head**, **tail**, **....**

Commande non filtres **: echo**, **ls**, **ps...**

- En annexe : liste de commandes utiles pour monitorer et gérer les processus
- En TD : gestion des processus – les bases

Annexes : liste de commandes utiles pour la gestion des processus

Récupérer le PID de la session shell courante

Le **PID** du shell courant est stocké dans une pseudo-variable spéciale que l'on appelle « **$** ». On peut le consulter grace à : **echo $$**

Le 1er "**$**" définit le contenu de la pseudo- variable. Le second "**$**" correspond à la variable stockant le PID du Shell courant.

La commande **ps**

La commande **ps** permet de visualiser les processus que lancés. Il y a plein d'options possible → **man ps**

prompt> **echo $$**
527
prompt> **cmd1 &**
prompt>
prompt> **ps**
PID TTY TIME COMMAND
527 ttyp4 1:70 -ksh
536 ttyp4 0:30 cmd1
559 ttyp4 0:00 ps
prompt>

PID identifie le processus,
TTY est le numéro du terminal associé,
TIME est le temps cumulé d'exécution du processus,
COMMAND est le nom du fichier correspondant au programme exécuté par le processus.

La commande ps – les options

Sans option, la commande concerne les processus associés au terminal depuis lequel elle est lancée.

ps	# liste des processus du shell courant
ps -ef	# liste de tous les processus
ps -ef | grep firefox	# Firefox est-il actif ?
ps -aux	# affiche les ressources utilisées
ps -u root	# les processus associés à un UID donné

L'option « **--forest** » permet de d'afficher en supplément l'arborescence des processus.

La commande type

type ***commande*** **...** donne le chemin absolu du fichier exécuté lorsque vous tapez ***commande***. Sinon, indique que la commande est interne au shell.

Exemples :

prompt> **type find pg**

find is /bin/find
pg is /usr/bin/pg

prompt> **type umask**
umask is a shell builtin
umask est une primitive du shell

- Commandes : **top**, **htop**, **jobs**

La commande top

Affiche **en temps réel** les processus qui consomment le plus de ressources systèmes. Dans les premières lignes, elle affiche des informations globales sur le système (charge, mémoire, nombre de processus, ...).

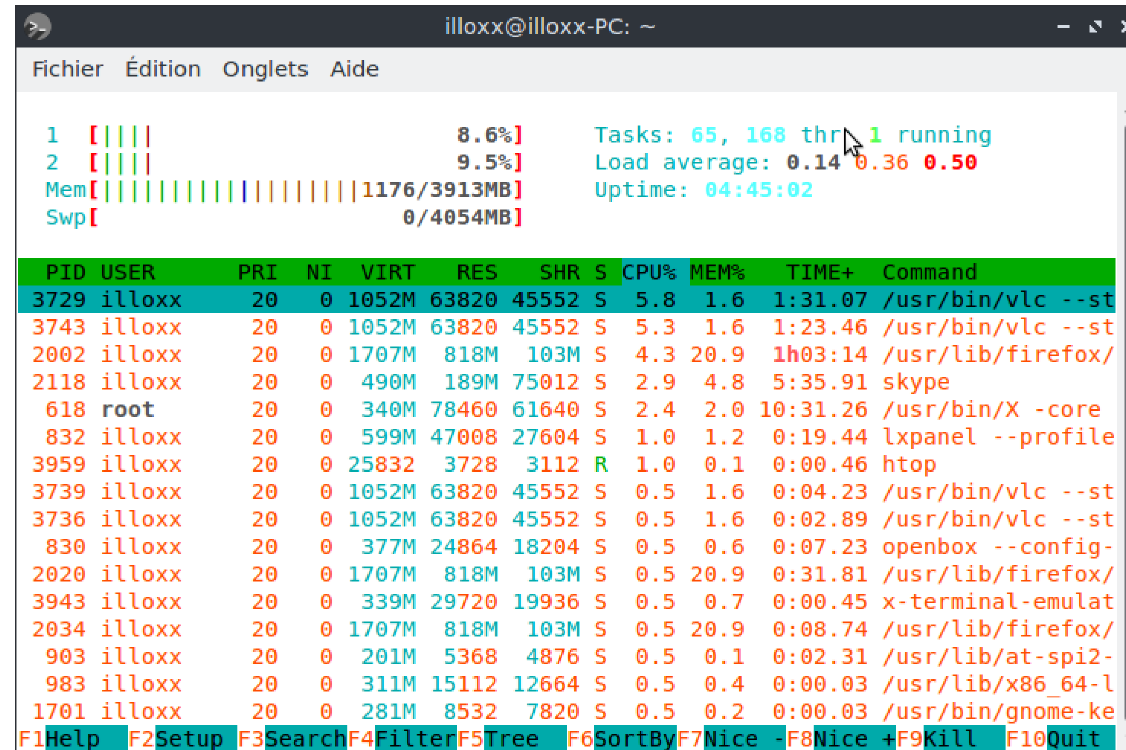
```
top - 11:14:18 up  1:02,  1 user,  load average: 0,05, 0,09, 0,08
Tasks: 209 total,   1 running, 208 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0,1 us,  0,2 sy,  0,0 ni, 99,8 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
MiB Mem :   7933,1 total,   6381,5 free,    797,0 used,    754,6 buff/cache
MiB Swap:   2048,0 total,   2048,0 free,      0,0 used.   6866,3 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    607 systemd+  20   0   16004   6300   5472 S   0,3   0,1   0:03.81 systemd+
   1327 jfa       20   0 4053620 269836 129020 S   0,3   3,3   0:37.19 gnome-s+
   2104 jfa       20   0  227344   2432   2072 S   0,3   0,0   0:06.34 VBoxCli+
   2207 jfa       20   0  563848  54248  41684 S   0,3   0,7   0:02.81 gnome-t+
   2718 jfa       20   0 2819336  64620  49212 S   0,3   0,8   0:00.68 gjs
      1 root      20   0  167916  12340   8576 S   0,0   0,2   0:01.75 systemd
      2 root      20   0       0      0      0 S   0,0   0,0   0:00.00 kthreadd
      3 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 rcu_par+
      5 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 netns
      6 root      20   0       0      0      0 I   0,0   0,0   0:01.01 kworker+
      7 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 kworker+
      9 root       0 -20       0      0      0 I   0,0   0,0   0:00.06 kworker+
     10 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 mm_perc+
     11 root      20   0       0      0      0 I   0,0   0,0   0:00.00 rcu_tas+
     12 root      20   0       0      0      0 I   0,0   0,0   0:00.00 rcu_tas+
     13 root      20   0       0      0      0 I   0,0   0,0   0:00.00 rcu_tas+
```

https://manpages.ubuntu.com/manpages/xenial/fr/man1/top.1.html

La commande htop

Similaire à top, mais interface un peu plus évoluée



https://doc.ubuntu-fr.org/htop

La commande jobs

La commande **jobs** est une commande des systèmes d'exploitation Unix et Unix-like pour lister les processus lancés ou suspendus en arrière-plan.

Elle **liste es processus en cours d'exécution** ainsi que **leur état** : **running** ou **stopped** ou **done**.

Syntaxe générale **jobs [option] [jobID]**

Exemple :

```
$ nano f1 &
$ firefox &
$ jobs
[1]-  Stopped                nano f1
[2]+  Running                firefox &
$
```

La commande **fg** est la commande qui permet de remettre un processus au premier plan (**f**ore**g**round)

Syntaxe générale : **fg [options] %[jobID]**

Le **jobID** est le numéro fournit par la commande **jobs**

Exemple :

```
$ jobs
[1]-  Stopped        nano f1
[2]+  Running        firefox &
$ fg %2
```

→ Affiche la fenêtre Firefox

La commande bg

La commande **bg** est la commande qui permet de remettre un processus au arrière plan (**b**ack**g**round)

Syntaxe générale : bg [options] %[jobID]

Le **jobID** est le numéro fournit par la commande **jobs**

Exemple :

$ **jobs**
[1]- Stopped nano f1
[2]+ Running firefox &
$ **bg %2**
→ Remet la fenêtre Firefox en arrière plan !

Les commandes pour tuer des processus

- **kill**, **killall**

La commande kill

kill envoie un **signal** à un (des) processus ou groupes de processus spécifiés, les obligeant à agir en fonction du signal. Lorsque le signal n'est pas spécifié, il est défini par défaut sur -**15 (-TERM)**.

Syntaxe générale : **kill [option] [PID]**

Les signaux habituellement utilisés :

- **1 (HUP)** - Reload a process.
- **9 (KILL)** - Kill a process.
- **15 (TERM)** - Gracefully stop a process.

kill -l liste les signaux disponibles

Exemple :

```
$ ps
  PID TTY          TIME CMD
 2226 pts/0    00:00:00 bash
 3614 pts/0    00:00:00 vi
 3617 pts/0    00:00:00 ps
$ kill -15 3614
$ ps
  PID TTY          TIME CMD
 2226 pts/0    00:00:00 bash
 3635 pts/0    00:00:00 ps
[1]+  Killed  vi toto
$
```

La commande killall

La commande **killall** est similaire à **kill** mais pour tous les processus qui exécutent une commande spécifique : elle envoie un signal à tous les processus ou groupes de processus dont le nom est spécifié, les obligeant à agir en fonction du signal. Lorsque le signal n'est pas spécifié, il est défini par défaut sur **-15 (-TERM)**.

Syntaxe générale : killall [option] Processus

Les signaux les plus communs :

1 (HUP) - Reload a process.
9 (KILL) - Kill a process.
15 (TERM) - Gracefully stop a process.

killall -l liste les signaux disponibles