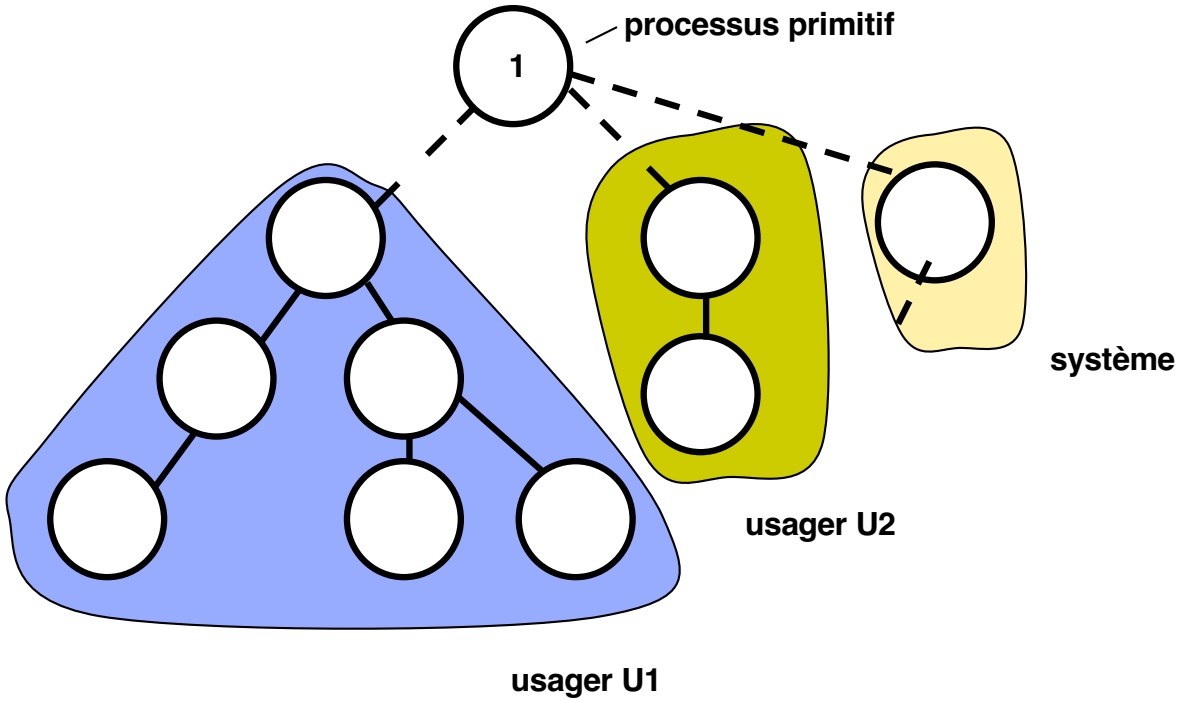
Processus

Vania Marangozova
Université Grenoble Alpes
2023-2024

vania.marangozova@imag.fr

Cours basé sur les transparents de
Sacha Krakowiak et Renaud Lachaize

Hiérarchie de processus dans Unix

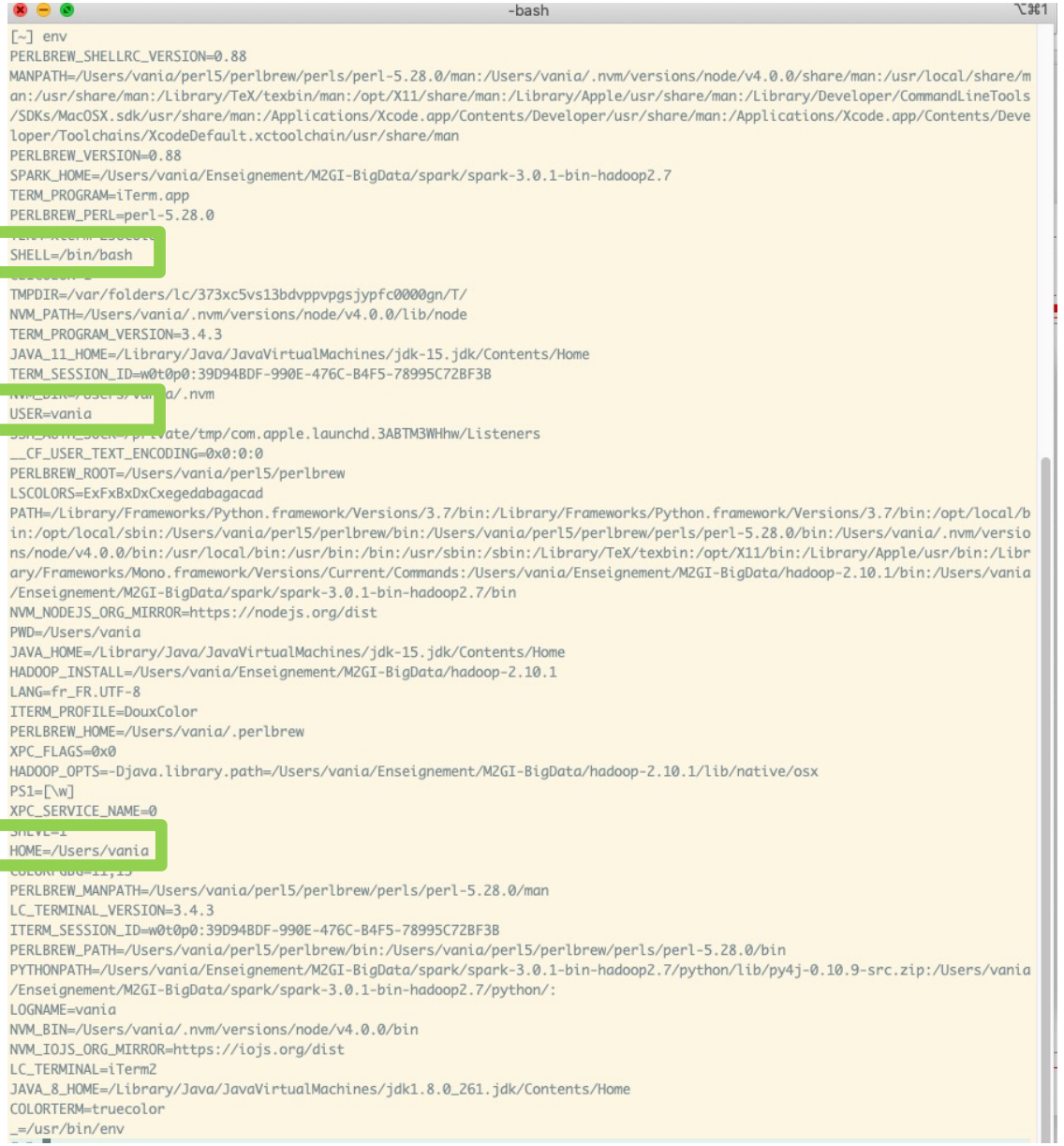


- **Fonctions utiles**
  - `getppid()` : obtenir le numéro du père
  - `getuid()` : obtenir le numéro d'usager (auquel appartient le processus)

Environnement d'un processus (1)

- **Dans Unix, un processus a accès à un certain nombre de variables qui constituent son environnement.**
  - faciliter la tâche de l'utilisateur en évitant d'avoir à redéfinir tout le contexte du processus (nom de l'utilisateur, de la machine, terminal par défaut, etc.)
  - personnaliser différents éléments comme le chemin de recherche des fichiers, le répertoire de base (home), le shell utilisé, etc.
- **Certaines variables sont prédéfinies dans le système. L'utilisateur peut les modifier, et peut aussi créer ses propres variables d'environnement.**
  - La commande **env** (sans paramètres) affiche l'environnement courant
- **Pour attribuer la valeur <val> à la variable VAR,**
  - Shell tcsh : **setenv VAR <val>**
  - Shell bash : **export VAR=<val>**
- **La commande echo $VAR affiche la valeur courante de la variable VAR**

Exemple


-bash
⌥⌘1
[~] env
PERLBREW_SHELLRC_VERSION=0.88
MANPATH=/Users/vania/perl5/perlbrew/perls/perl-5.28.0/man:/Users/vania/.nvm/versions/node/v4.0.0/share/man:/usr/local/share/man:/usr/share/man:/Library/TeX/texbin/man:/opt/X11/share/man:/Library/Apple/usr/share/man:/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/share/man:/Applications/Xcode.app/Contents/Developer/usr/share/man:/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/share/man
PERLBREW_VERSION=0.88
SPARK_HOME=/Users/vania/Enseignement/M2GI-BigData/spark/spark-3.0.1-bin-hadoop2.7
TERM_PROGRAM=iTerm.app
PERLBREW_PERL=perl-5.28.0
SHELL=/bin/bash
TMPDIR=/var/folders/lc/373xc5vs13bdvppvpgsjypfc0000gn/T/
NVM_PATH=/Users/vania/.nvm/versions/node/v4.0.0/lib/node
TERM_PROGRAM_VERSION=3.4.3
JAVA_11_HOME=/Library/Java/JavaVirtualMachines/jdk-15.jdk/Contents/Home
TERM_SESSION_ID=w0t0p0:39D94BDF-990E-476C-B4F5-78995C72BF3B
USER=vania
__CF_USER_TEXT_ENCODING=0x0:0:0
PERLBREW_ROOT=/Users/vania/perl5/perlbrew
LSCOLORS=ExFxBxDxCxegedabagacad
PATH=/Library/Frameworks/Python.framework/Versions/3.7/bin:/Library/Frameworks/Python.framework/Versions/3.7/bin:/opt/local/bin:/opt/local/sbin:/Users/vania/perl5/perlbrew/bin:/Users/vania/perl5/perlbrew/perls/perl-5.28.0/bin:/Users/vania/.nvm/versions/node/v4.0.0/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/TeX/texbin:/opt/X11/bin:/Library/Apple/usr/bin:/Library/Frameworks/Mono.framework/Versions/Current/Commands:/Users/vania/Enseignement/M2GI-BigData/hadoop-2.10.1/bin:/Users/vania/Enseignement/M2GI-BigData/spark/spark-3.0.1-bin-hadoop2.7/bin
NVM_NODEJS_ORG_MIRROR=https://nodejs.org/dist
PWD=/Users/vania
JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-15.jdk/Contents/Home
HADOOP_INSTALL=/Users/vania/Enseignement/M2GI-BigData/hadoop-2.10.1
LANG=fr_FR.UTF-8
ITERM_PROFILE=DouxColor
PERLBREW_HOME=/Users/vania/.perlbrew
XPC_FLAGS=0x0
HADOOP_OPTS=-Djava.library.path=/Users/vania/Enseignement/M2GI-BigData/hadoop-2.10.1/lib/native/osx
PS1=[\w]
XPC_SERVICE_NAME=0
HOME=/Users/vania
PERLBREW_MANPATH=/Users/vania/perl5/perlbrew/perls/perl-5.28.0/man
LC_TERMINAL_VERSION=3.4.3
ITERM_SESSION_ID=w0t0p0:39D94BDF-990E-476C-B4F5-78995C72BF3B
PERLBREW_PATH=/Users/vania/perl5/perlbrew/bin:/Users/vania/perl5/perlbrew/perls/perl-5.28.0/bin
PYTHONPATH=/Users/vania/Enseignement/M2GI-BigData/spark/spark-3.0.1-bin-hadoop2.7/python/lib/py4j-0.10.9-src.zip:/Users/vania/Enseignement/M2GI-BigData/spark/spark-3.0.1-bin-hadoop2.7/python/:
LOGNAME=vania
NVM_BIN=/Users/vania/.nvm/versions/node/v4.0.0/bin
NVM_IOJS_ORG_MIRROR=https://iojs.org/dist
LC_TERMINAL=iTerm2
JAVA_8_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home
COLORTERM=truecolor
_=/usr/bin/env

Environnement d'un processus (2)

On peut aussi consulter et modifier les variables d'environnement par programme

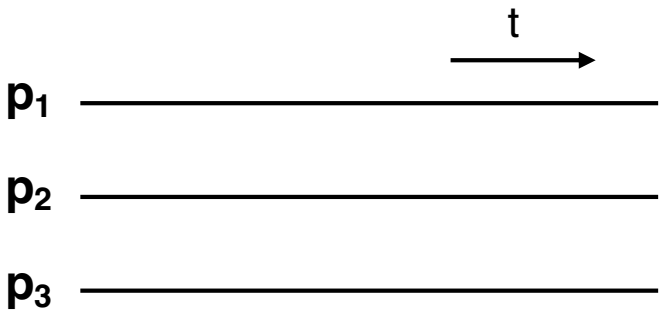
```
#include <stdlib.h>
char * getenv (const char *nom);
int putenv (const char *chaine);
int setenv (const char *nom, const char *valeur, int écraser);
int unsetenv (const char *nom);
```

Exemples :

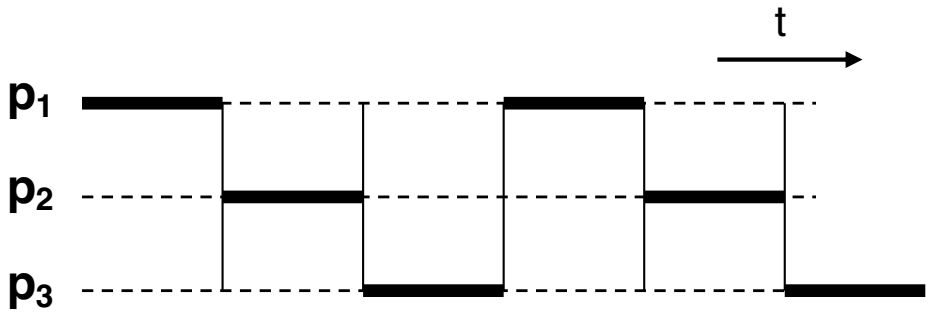
```
char *var = getenv("USER");
if (var != NULL) printf("utilisateur: %s", var);

putenv("MAVARIABLE=toto");
setenv("MAVARIABLE", "toto", 1);

unsetenv("MAVARIABLE"); // équivalent à putenv("MAVARIABLE") !!!
```

Réalisation des processus

- **Processus = mémoire virtuelle + flot d'exécution (processeur virtuel)**
  - Ces deux ressources sont fournies par le système d'exploitation, qui alloue les ressources physiques de la machine
  - La gestion de la mémoire n'est pas étudiée ici en détail (cf cours ultérieur dans cette UE et cours de M1)
  - L'allocation de processeur est réalisée par multiplexage (allocation successive aux processus pendant une tranche de temps fixée)

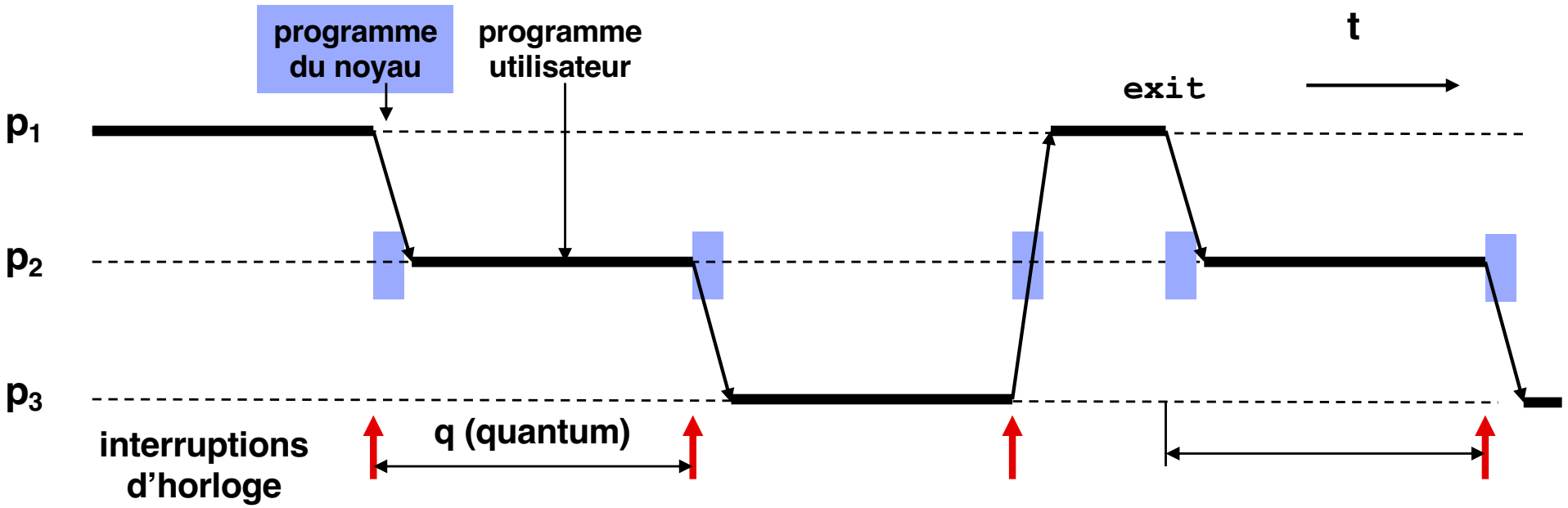


vue abstraite
t = temps logique

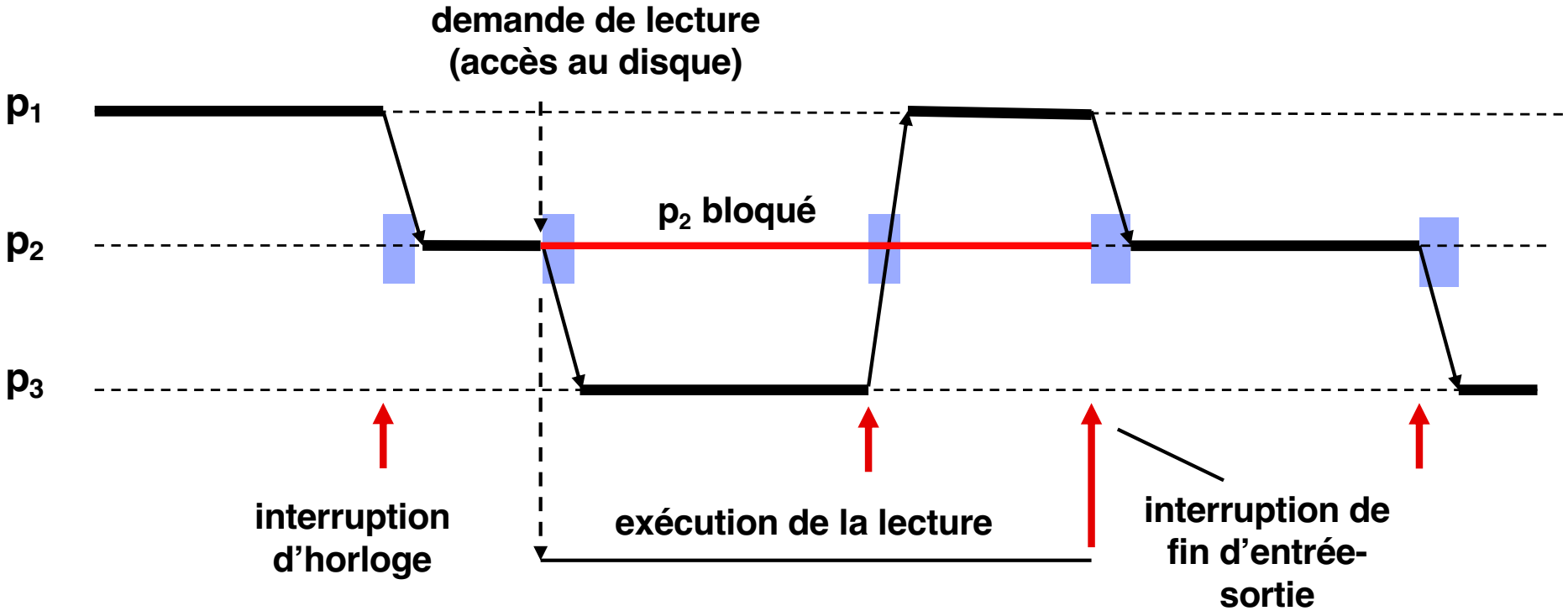


vue concrète (très simplifiée)
t = temps physique

Allocation du processeur aux processus (1)

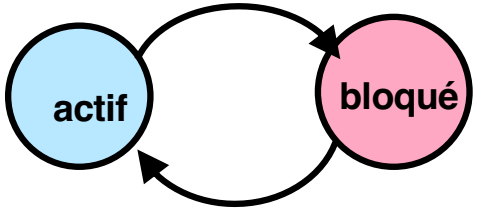


Allocation du processeur aux processus (2)



États d'un processus

États logiques

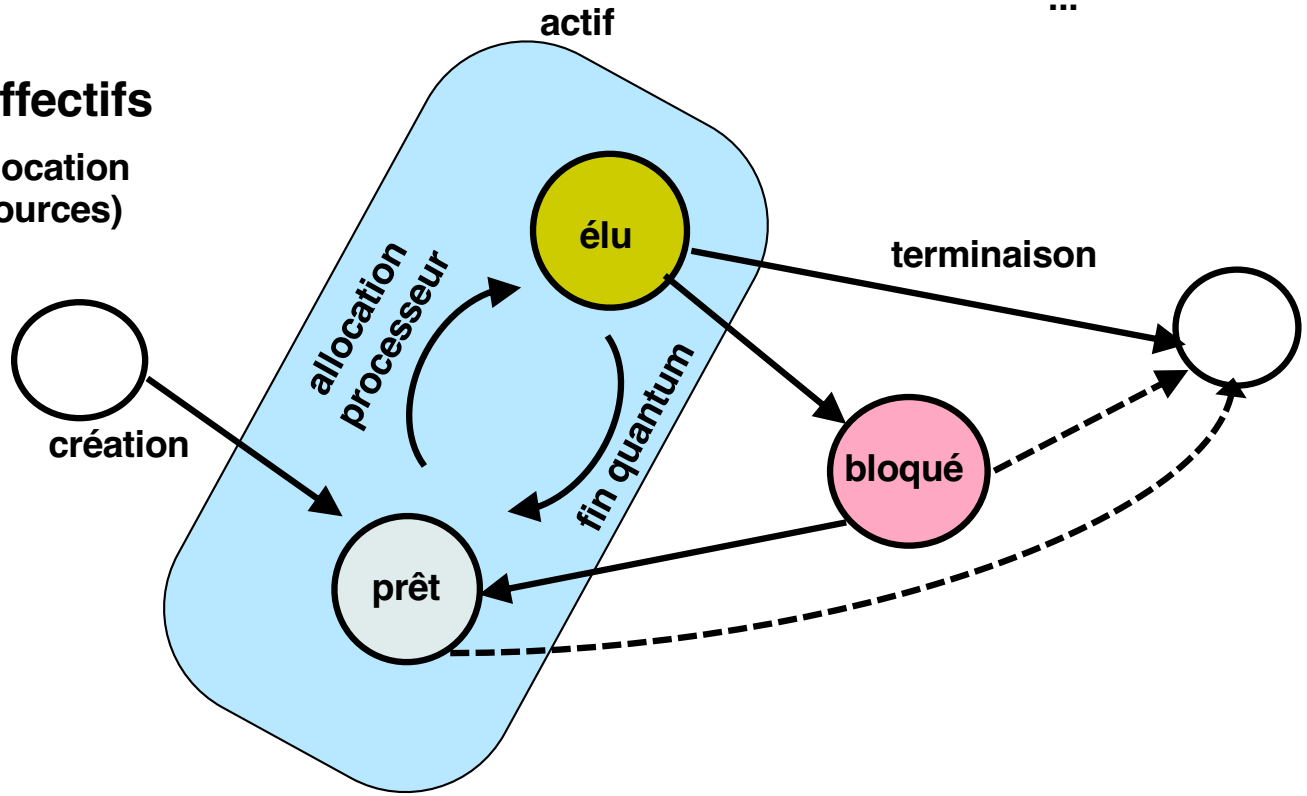


causes de blocage :

entrée-sortie
attente d'un signal
attente de la terminaison d'un fils
endormissement temporaire (sleep)
...

États effectifs

(avec allocation de ressources)



PAUSE ?

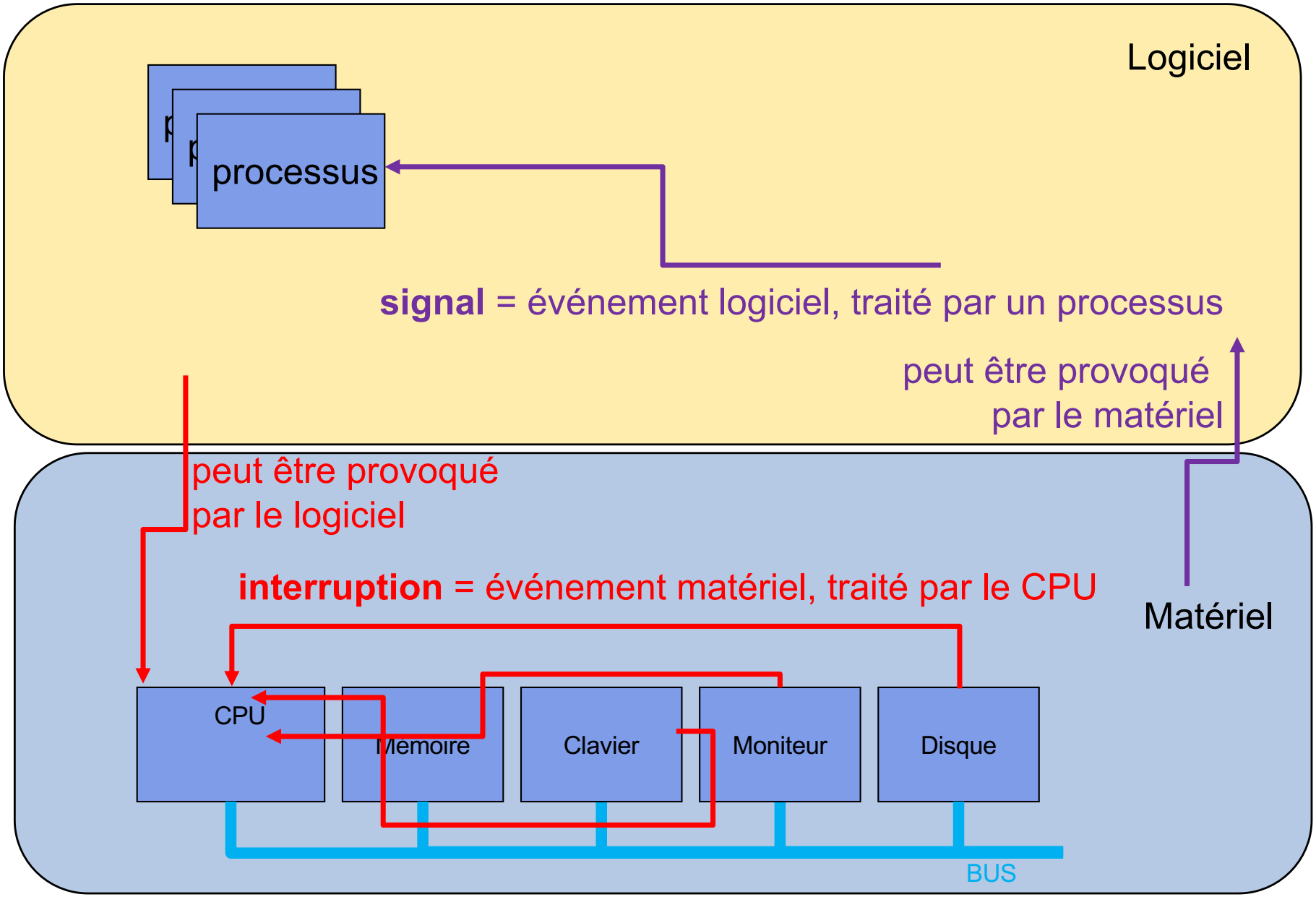
Signaux

- **Un signal est un événement asynchrone destiné à un (ou plusieurs) processus. Un signal peut être émis par un processus ou par le système d'exploitation.**

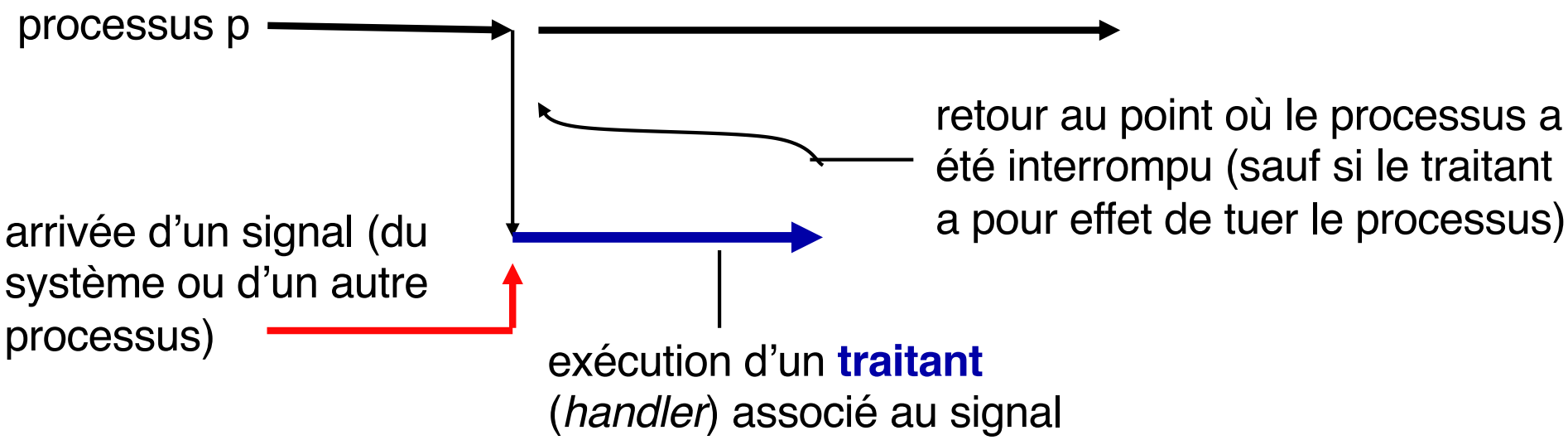
- **Un signal est analogue à une interruption : un processus destinataire réagit à un signal en exécutant un programme de traitement, ou traitant (*handler*). La différence est qu'une interruption s'adresse à un processeur alors qu'un signal s'adresse à un processus. Certains signaux traduisent d'ailleurs la réception d'une interruption (voir plus loin).**

- **Les signaux sont un mécanisme de bas niveau. Ils doivent être manipulés avec précaution car leur usage recèle des pièges (en particulier le risque de perte de signaux). Ils sont néanmoins utiles lorsqu'on doit contrôler l'exécution d'un ensemble de processus (exemple : le *shell*) ou que l'on traite des événements liés au temps.**

Signaux


Logiciel
processus
signal = événement logiciel, traité par un processus
peut être provoqué
par le matériel
peut être provoqué
par le logiciel
interruption = événement matériel, traité par le CPU
Matériel
CPU
Mémoire
Clavier
Moniteur
Disque
BUS

Fonctionnement des signaux



Points à noter (seront précisés plus loin) :

- Il existe différents signaux, chacun étant identifié par un **nom symbolique** (ce nom représente un entier)
- Chaque signal est associé à un **traitant par défaut**
- Un signal peut être **ignoré** (le traitant est vide)
- Le traitant d'un signal peut être changé (sauf pour 2 signaux particuliers)
- Un signal peut être **bloqué** (il n'aura d'effet que lorsqu'il sera débloqué)
- Les signaux **ne sont pas mémorisés** (détails plus loin)

Quelques exemples de signaux

Nom symbolique	Événement associé	Défaut
SIGINT	Frappe du caractère <control-C>	terminaison
SIGTSTP	Frappe du caractère <control-Z>	suspension
SIGKILL	Signal de terminaison	terminaison
SIGSTOP	Signal de suspension	suspension
SIGSEGV	Violation de protection mémoire	terminaison +core dump
SIGALRM	Fin de temporisation (alarm)	terminaison
SIGCHLD	Terminaison d'un fils	ignoré
SIGUSR1	Signal émis par un processus utilisateur	terminaison
SIGUSR2	Signal émis par un processus utilisateur	terminaison
SIGCONT	Continuation d'un processus stoppé	reprise

Notes. Les signaux SIGKILL et SIGSTOP ne peuvent pas être bloqués ou ignorés, et leur traitant ne peut pas être changé (pour des raisons de sécurité).
Utiliser toujours les noms symboliques, non les valeurs (exemple : SIGINT = 2, etc.) car ces valeurs peuvent changer d'un Unix à un autre. Inclure <signal.h>. Voir man 7 signal

États d'un signal

Un signal

- **est envoyé à un processus destinataire et**
- **reçu par ce processus**
- **le signal est pendant, tant qu'il n'a pas été pris en compte par le destinataire**
- **le signal est dit traité, lorsqu'il est pris en compte (exécution du traitant)**

Qu'est-ce qui empêche que le signal soit immédiatement traité dès qu'il est reçu ?

- Le signal peut être bloqué, ou masqué (c'est à dire retardé) par le destinataire. Il est délivré dès qu'il est débloqué
- En particulier, un signal est bloqué pendant l'exécution du traitant d'un signal du même type ; il reste bloqué tant que ce traitant n'est pas terminé

Point important : il ne peut exister qu'un seul signal pendant d'un type donné (il n'y a qu'un bit par signal pour indiquer les signaux de ce type qui sont pendants). S'il arrive un autre signal du même type, il est perdu.

Envoi d'un signal

Un processus peut envoyer un signal à un autre processus. Pour cela, il utilise la primitive `kill` (appelée ainsi pour des raisons historiques ; un signal ne tue pas forcément son destinataire).

Utilisation : `kill(pid_t pid, int sig)`

Effet : Soit `p` le numéro du processus émetteur du signal
Le signal de numéro `sig` est envoyé au(x) processus désigné(s) par `pid` :

- si `pid > 0` le signal est envoyé au processus de numéro `pid`
- si `pid = 0` le signal est envoyé à tous les processus du même groupe que `p`
- si `pid = -1` le signal est envoyé à tous les processus
- si `pid < 0` le signal est envoyé à tous les processus du groupe `-pid`

Restrictions : un processus (sauf s'il a les droits de `root`) n'est autorisé à envoyer un signal qu'aux processus ayant le même `uid` (identité d'utilisateur) que lui.

Le processus de numéro 1 ne peut pas recevoir certains signaux (notamment SIGKILL). Étant donné son rôle particulier, il est protégé pour assurer la sécurité et la stabilité du système.

Quelques exemples d'utilisation des signaux

On va donner plusieurs exemples d'utilisation des signaux

- Interaction avec le travail de premier plan : SIGINT, SIGTSTP
- Signaux de temporisation : SIGALRM
- Relations père-fils : SIGCHLD

Dans tous ces cas, on aura besoin de redéfinir le traitant associé à un signal (c'est-à-dire lui associer un traitant autre que le traitant par défaut). On va donc d'abord montrer comment le faire.

Redéfinir le traitant associé à un signal

On utilise la structure suivante :

```
struct sigaction {
    void (*sa_handler)() ;      pointeur sur traitant
    sigset_t sa_mask;           autres signaux à bloquer
    int sa_flags;               options
}
```

et la primitive :

```
int sigaction(int sig, const struct sigaction *newaction,
    struct sigaction *oldaction);
```

Pour notre pratique, nous utiliserons une primitive `Signal` qui "enveloppe" ces éléments et qui est fournie dans le programme `csapp.c` :

```
#include <signal.h>
typdef void handler_t (int)    un paramètre entier, ne renvoie rien
handler_t *Signal(int signum, handler_t *handler)
```

Associe le traitant `handler` au signal de numéro `signum`

Exemple 1 : traitement d'une interruption du clavier

Il suffit de redéfinir le traitant de SIGINT (qui par défaut tue le processus interrompu)

test-int.c

```
#include "csapp.h"
void handler(int sig) {    /* nouveau traitant    */
  printf("signal SIGINT reçu !\n");
  exit(0);
}

int main() {
  Signal(SIGINT, handler); /* installe le traitant */
  pause () ;               /* attend un signal     */
  exit(0);
}
```

```
<unix>./test-int
    => frappe de control-C
signal SIGINT reçu !
<unix>
```

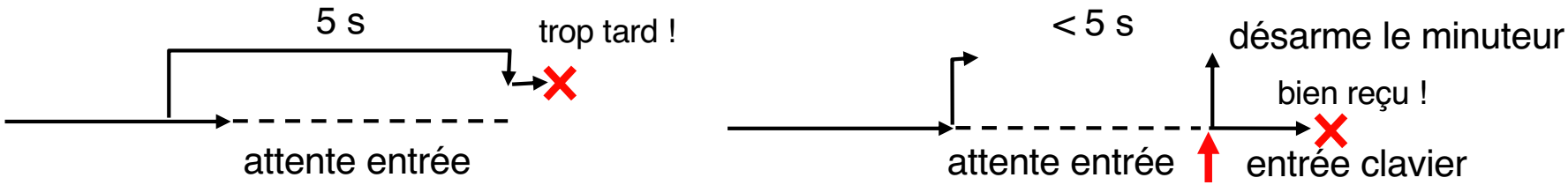
Exercice : modifier ce programme pour qu'il poursuive son exécution après la frappe de *control-C* (il pourra être interrompu à nouveau)

Exemple 2 : utilisation de la temporisation

- **La primitive `alarm` provoque l'envoi du signal SIGALRM après environ `nbSec` secondes; annulation avec `alarm(0)`**

```
#include <unistd.h>
unsigned int alarm(unsigned int nbSec)
```

```
#include "csapp.h"
void handler(int sig) {    /* nouveau traitant    */
  printf("trop tard !\n");
  exit(0);
}
int main() {
  int reponse;
  Signal(SIGALRM, handler); /* installe le traitant */
  printf("entrer un nombre avant 5 sec. : ") ; alarm(5);
  scanf("%d", &reponse); alarm(0); printf("bien reçu !\n");
  exit (0),
}
```



Exemple 3 : synchronisation père-fils

- **Lorsqu'un processus se termine ou est suspendu, le système envoie automatiquement un signal SIGCHLD à son père.**
  - Le traitement par défaut consiste à ignorer ce signal.
- **On peut prévoir un traitement spécifique en associant un nouveau traitant à SIGCHLD**
- **Application : lorsqu'un processus crée un grand nombre de fils (par exemple un *shell* crée un processus pour traiter chaque commande), il doit prendre en compte leur fin dès que possible pour éviter une accumulation de processus zombis (qui consomment de la place dans les tables du système).**

```
#include "csapp.h"
void handler(int sig) {          /* nouveau traitant          */
  pid_t pid; int statut;
  pid = waitpid(-1, &statut, 0); /* attend un fils quelconque */
  return;
}
int main() {
  Signal(SIGCHLD, handler);      /* installe le traitant      */
  ... <création d'un certain nombre de fils, sur demande> ...
  exit (0),
}
```

Gestion du masque de signaux

- **Masquer des signaux**
  - automatiquement lors du traitant, typiquement le signal lui-même
  - pas dans le traitant, dans le code en dehors du traitant

```
int main() {
  sigset_t s; // structure de données opaque permettant de
              // désigner un ou plusieurs types de signaux
  ...
  Sigemptyset(&s); //s <- ensemble vide
  Sigaddset(&s, SIGINT); // ajout de SIGINT dans l'ensemble s
  Sigprocmask(SIG_BLOCK, &s, NULL);
  // à ce stade là, le masquage de SIGINT est effectif
  ...
  Sigprocmask(SIG_UNBLOCK, &s, NULL);
  // à ce stade là, SIGINT est à nouveau démasqué
  ...
}
```

Terminaux, sessions et groupes sous Unix (1)

Pour bien comprendre le fonctionnement de certains signaux, il faut avoir une idée des notions de session et de groupes (qui seront revues plus tard à propos du *shell*).

- **En première approximation, une session est associée à un terminal**
  - donc au *login* d'un usager du système au moyen d'un *shell*.
  - Le processus qui exécute ce *shell* est le *leader* de la session.
- **Dans une session**
  - on peut avoir plusieurs groupes de processus correspondant à divers travaux en cours.
  - **le groupe interactif** (*foreground*, ou premier plan) avec lequel l'usager interagit via le terminal
  - **il peut exister plusieurs groupes d'arrière-plan**, qui s'exécutent en travail de fond
    - par exemple processus lancés avec &, appelés *jobs*
- **Seuls les processus du groupe interactif peuvent lire au terminal. D'autre part, les signaux SIGINT (frappe de control-C) et SIGTSTP (frappe de control-Z) s'adressent au groupe interactif et non aux groupes d'arrière-plan.**

Terminaux, sessions et groupes sous Unix (2)

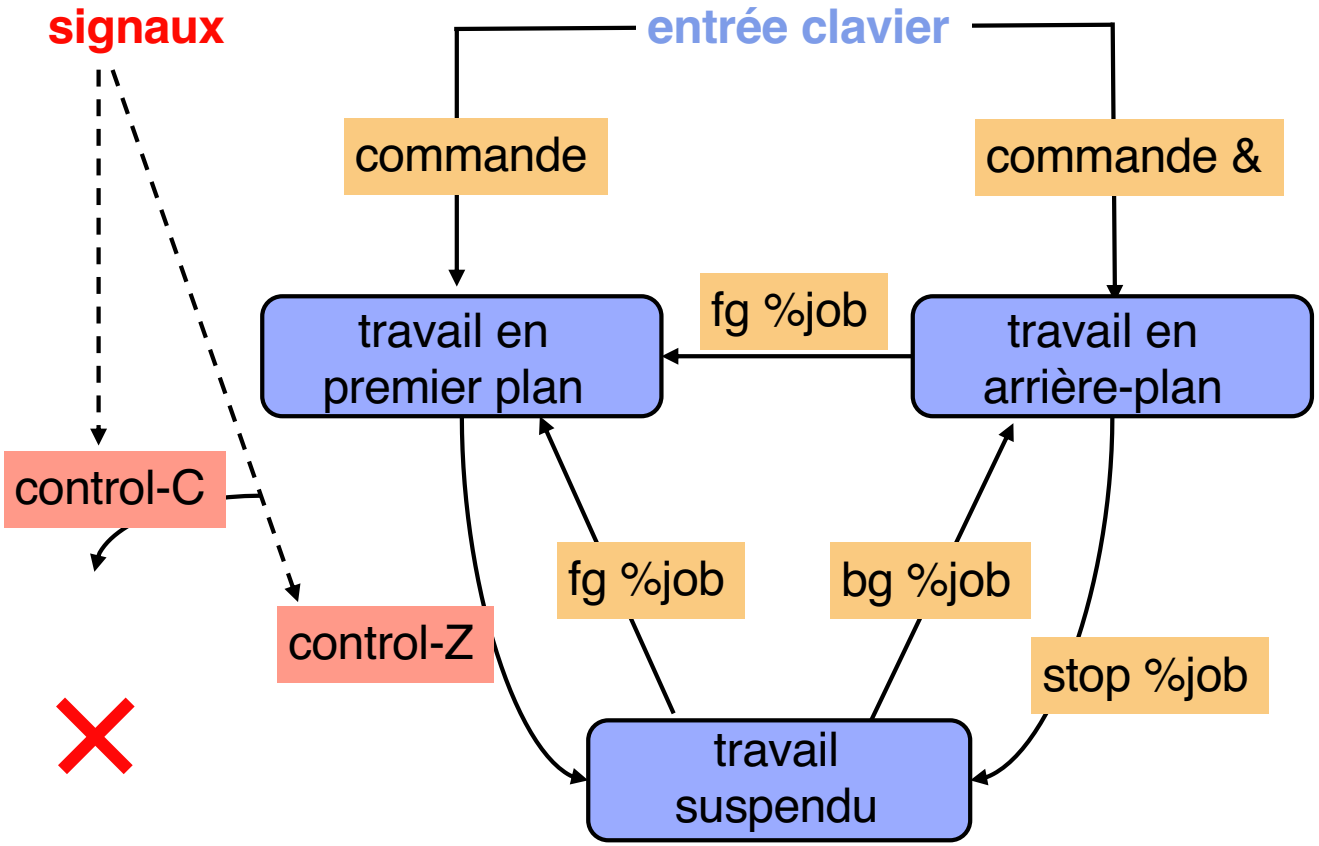
loop.c

```
int main() {
   printf"processus %d, groupe %d\n", getpid(), getpgrp());
   while(1) ;
}
```

```
<unix> loop & loop & ps
processus 10468, groupe 10468
[1]  10468
processus 10469, groupe 10469
[2] 10469
  PID TTY          TIME  CMD
 5691 pts/0  00:00:00  tcsh
10468 pts/0  00:00:00  loop
10469 pts/0  00:00:00  loop
10470 pts/0  00:00:00  ps
<unix>fg %1
loop
        =>frappe de control-Z
Suspended
<unix>jobs
[1] + Suspended          loop
[2] - Running            loop
```

```
<unix>bg %1
[1]  loop &
<unix>fg %2
loop
        => frappe de control-C
<unix>ps
  PID TTY          TIME  CMD
 5691 pts/0  00:00:00  tcsh
10468 pts/0  00:02:53  loop
10474 pts/0  00:00:00  ps
<unix>  => frappe de control-C
<unix>ps
  PID TTY          TIME  CMD
 5691 pts/0  00:00:00  tcsh
10468 pts/0  00:02:57  loop
10474 pts/0  00:00:00  ps
<unix>
```

États d'un travail (*job*)



Travail (*job*) = processus (ou groupe de processus) lancé par une commande au *shell*

Seuls le travail en premier plan peut recevoir des signaux du clavier. Les autres sont manipulés par des commandes

Vie des travaux