Processus (suite)

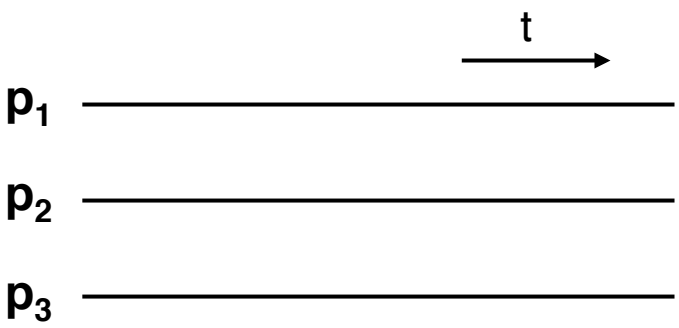
Réalisation des processus
Communication par signaux

Olivier Dalle
Université de Nice - Sophia Antipolis
http://deptinfo.unice.fr/
D'après le cours original de
Sacha Krakowiak
Université Joseph Fourier
Projet Sardes (INRIA et IMAG-LSR)

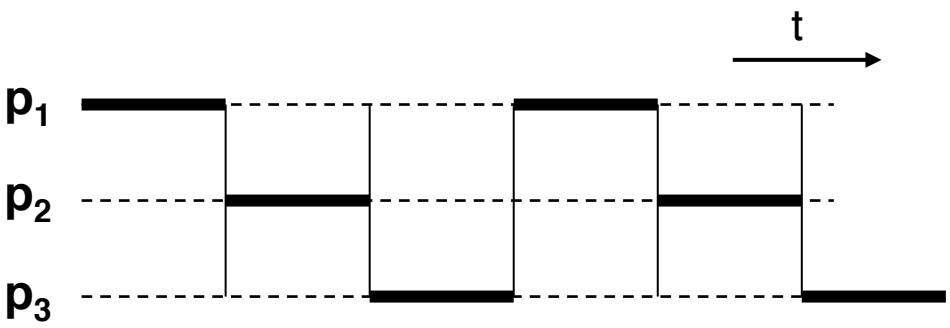
http://sardes.inrialpes.fr/~krakowia

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
  - L'allocation de mémoire n'est pas étudiée ici en détail (cf cours 5 et cours de M1) : nous décrivons seulement l'organisation interne de la mémoire virtuelle d'un processus dans Unix
  - L'allocation de processeur est réalisée par multiplexage (allocation successive aux processus pendant une tranche de temps fixée)

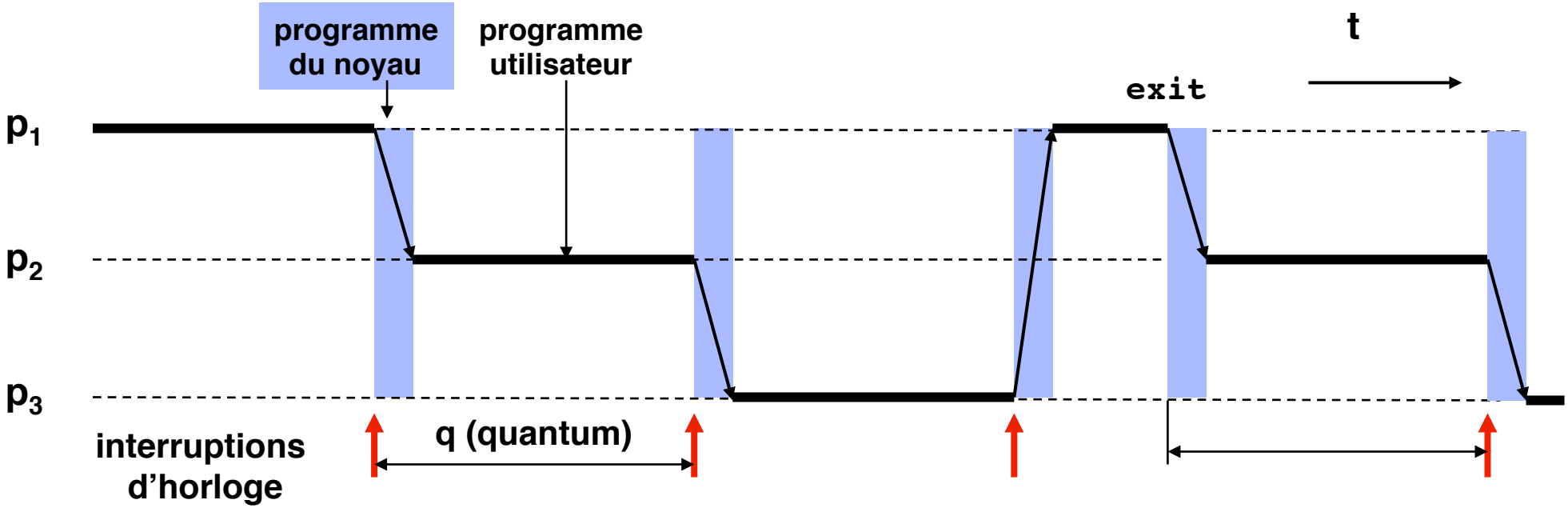


vue abstraite
t = temps logique



vue concrète (très simplifiée)
t = temps physique

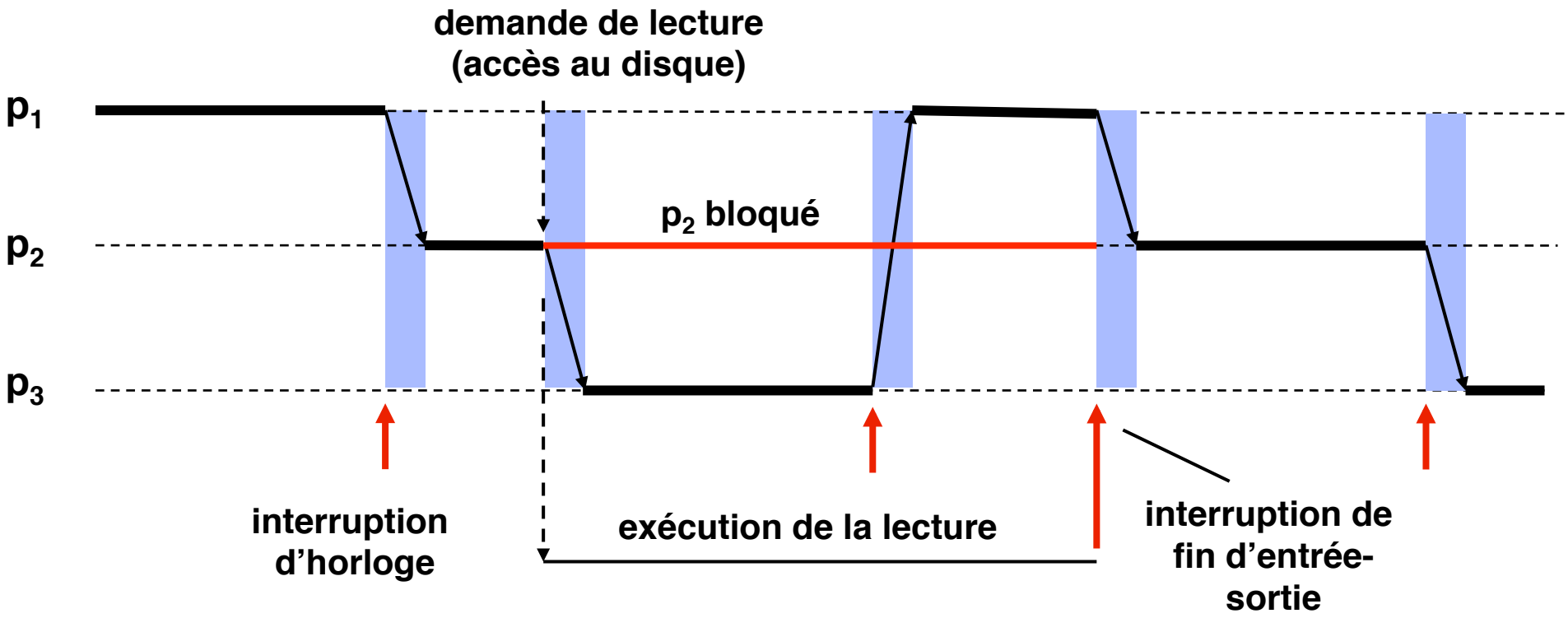
Allocation du processeur aux processus (1)



Le processeur est alloué par tranches de temps successives (quanta) aux processus prêts à s'exécuter (non bloqués). Dans la pratique, la valeur du quantum est d'environ 10 ms (temps d'exécution de quelques millions d'instructions sur un processeur à 1 GHz). La commutation entre processus est déclenchée par une interruption d'horloge.

La commutation entre processus prend un temps non nul (de l'ordre de 0,5 ms). Elle est réalisée par un programme du noyau appelé ordonnanceur (*scheduler*)

Allocation du processeur aux processus (2)

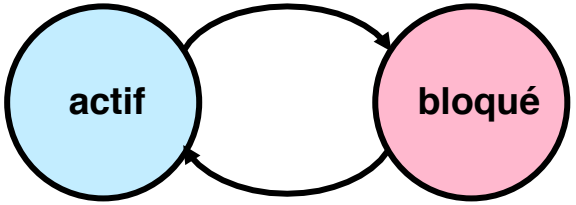


Lorsqu'un processus est bloqué (par exemple parce qu'il a demandé une entrée sortie, ou a appelé *sleep*), il doit libérer le processeur puisqu'il ne peut plus l'utiliser. Le processeur pourra lui être réalloué à la fin de sa période de blocage (souvent indiquée par une interruption)

Cette réallocation pourra se faire soit immédiatement (comme sur la figure) ou plus tard (après fin de quantum), selon la politique choisie.

États d'un processus

États logiques

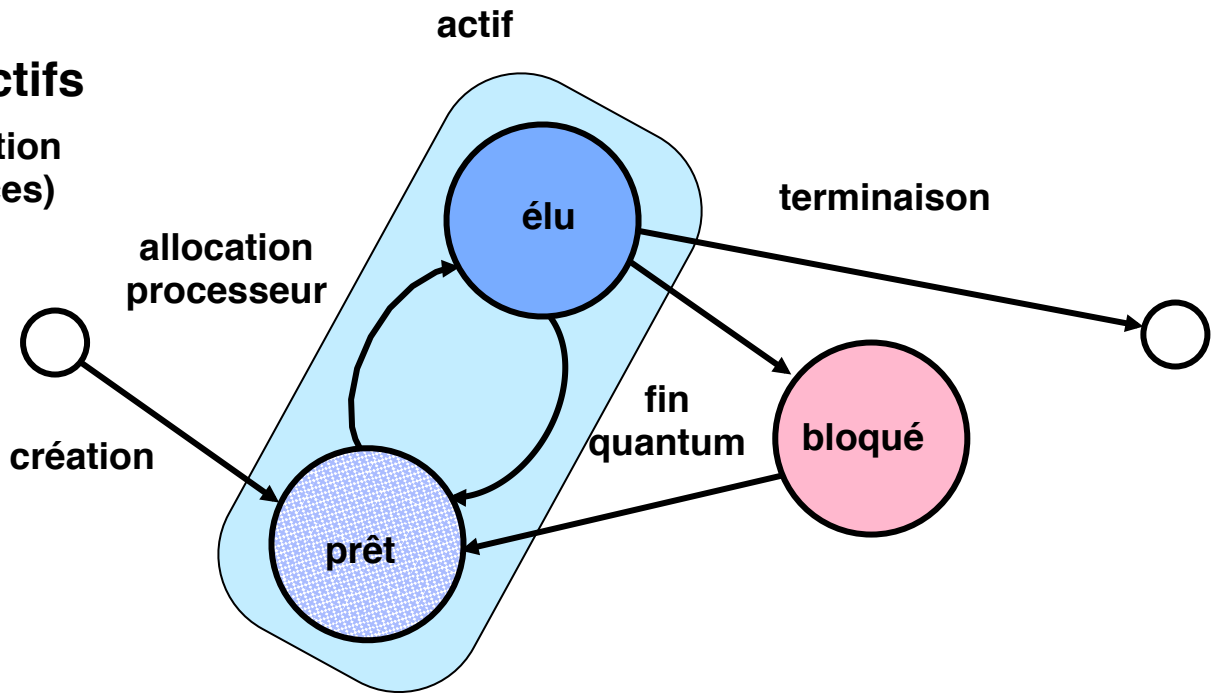


causes de blocage :

entrée-sortie
attente d'un signal
(entre autres `wait`, `pause`, `sleep`,...)

États effectifs

(avec allocation de ressources)



Mécanisme d'allocation du processeur

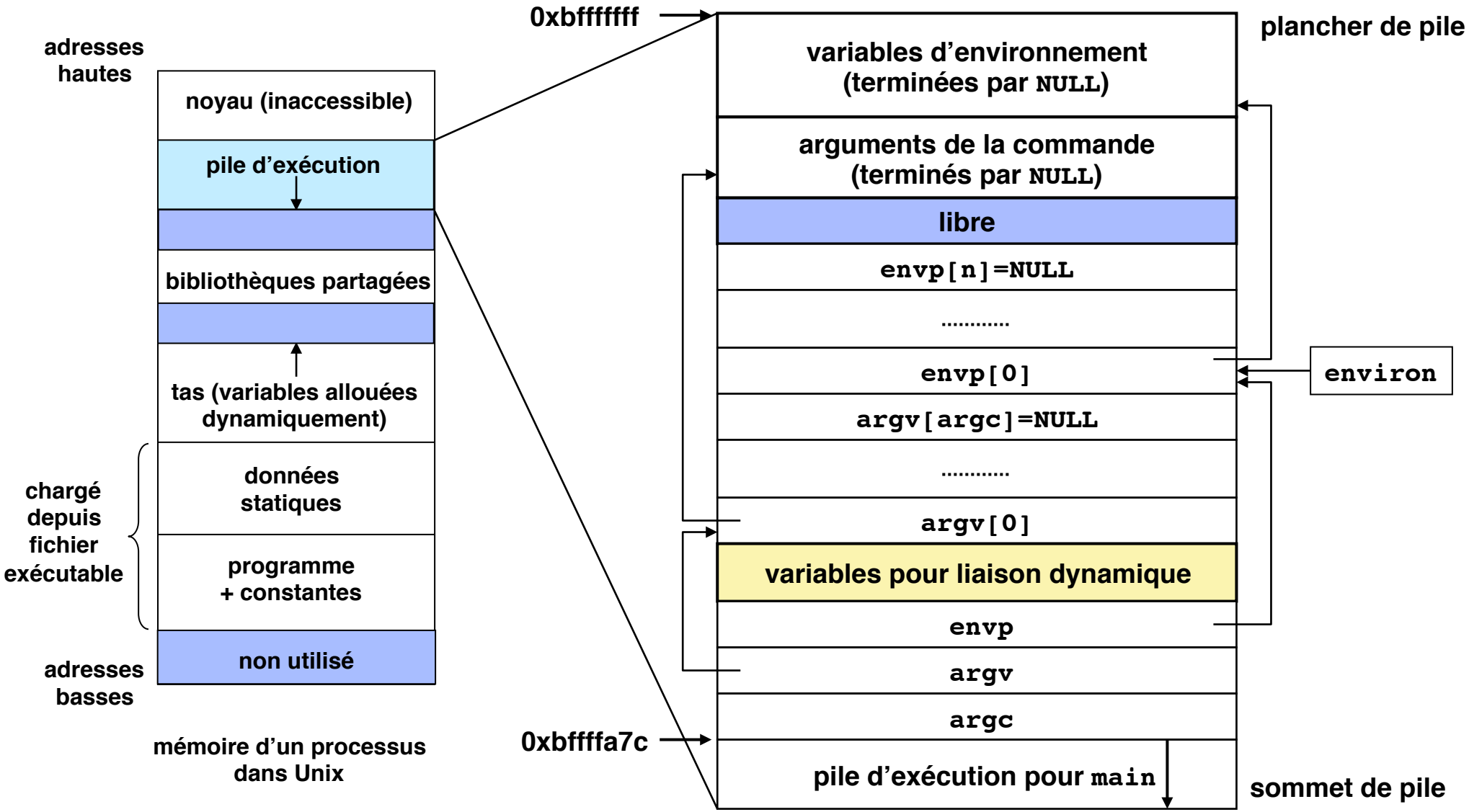
Que fait précisément le noyau du système lors de la commutation de processus ?

1. Déterminer le prochain processus élu (en général, les processus sont élus dans l'ordre d'arrivée, mais il peut y avoir des priorités). Le mécanisme utilise une file d'attente.

2. Réaliser l'allocation proprement dite. Deux étapes :

   a) Sauvegarder le contexte du processus élu actuel (contenu des registres programmables et des registres internes), pour pouvoir le retrouver ultérieurement, en vue de la prochaine allocation
   b) Restaurer le contexte du nouveau processus élu ; en particulier :
      - restaurer les structures de données qui définissent la mémoire virtuelle
      - charger les registres, puis le "mot d'état" du processeur, ce qui lance l'exécution du nouveau processus élu

Structure de la mémoire virtuelle d'un processus (en C)



Communication entre processus dans Unix

La **communication entre processus** (*Inter-Process Communication*, ou **IPC**) est l'un des aspects les plus importants (et aussi les plus délicats) de la programmation de systèmes.

Dans Unix, cette communication peut se faire de plusieurs manières différentes, que nous n'examinons pas toutes

- Communication asynchrone au moyen de **signaux** [suite de cette séance]
- Communication par **fichiers** ou par **tubes** (*pipes*, FIFOs) [à voir dans cours 6]
- Communication par **files de messages** [non étudié, voir cours de L3]
- Communication par **mémoire partagée** (sera un peu vu dans cours 5) et par **sémaphores** [non étudié, voir cours de L3]
- Communication par ***sockets***, utilisés dans les réseaux, mais aussi en local [à voir dans cours 7]

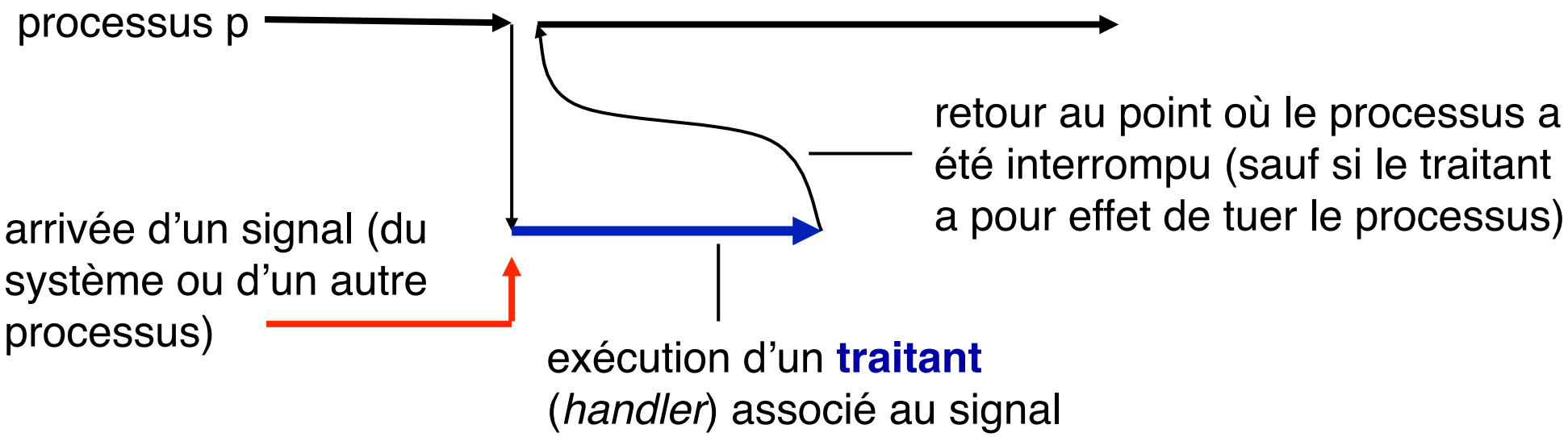
Signaux

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.

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**

Quelques exemples de signaux

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

Notes. Les signaux signal.SIGKILL et signal.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. Voir man 7 signal (en C) et module signal (en Python).

États d'un signal

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

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

Les signaux sont traités dans l'ordre croissant de leurs numéros

Structures internes associées aux signaux

Table pour chaque processus :

				1	2		NSIG–1
1	0/1	0/1	void (*)(int)	1	0/1	...	0/1
2	0/1	0/1	void (*)(int)	0/1	1	...	0/1
...							
NSIG–1	0/1	0/1	void (*)(int)	0/1	0/1	...	1

n° de signal — pendant — bloqué — pointeur vers traitant — masque temporaire (pendant l'exécution du traitant)

Quand un signal d'un type i donné est reçu, pendant[i]=1. Si alors bloqué[i]=0, le signal est traité et pendant[i] est remis à 0. Pendant l'exécution du traitant, un masque temporaire est placé (le signal i est automatiquement bloqué pendant l'exécution de son traitant, et d'autres signaux peuvent être bloqués)

Si un signal i arrive alors que pendant[i]=1, alors il est perdu

Terminaux, sessions et groupes en 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. Il existe au plus **un groupe interactif** (*foreground*, ou premier plan) avec lequel l'usager interagit via le terminal. Il peut aussi 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 en Unix (2)

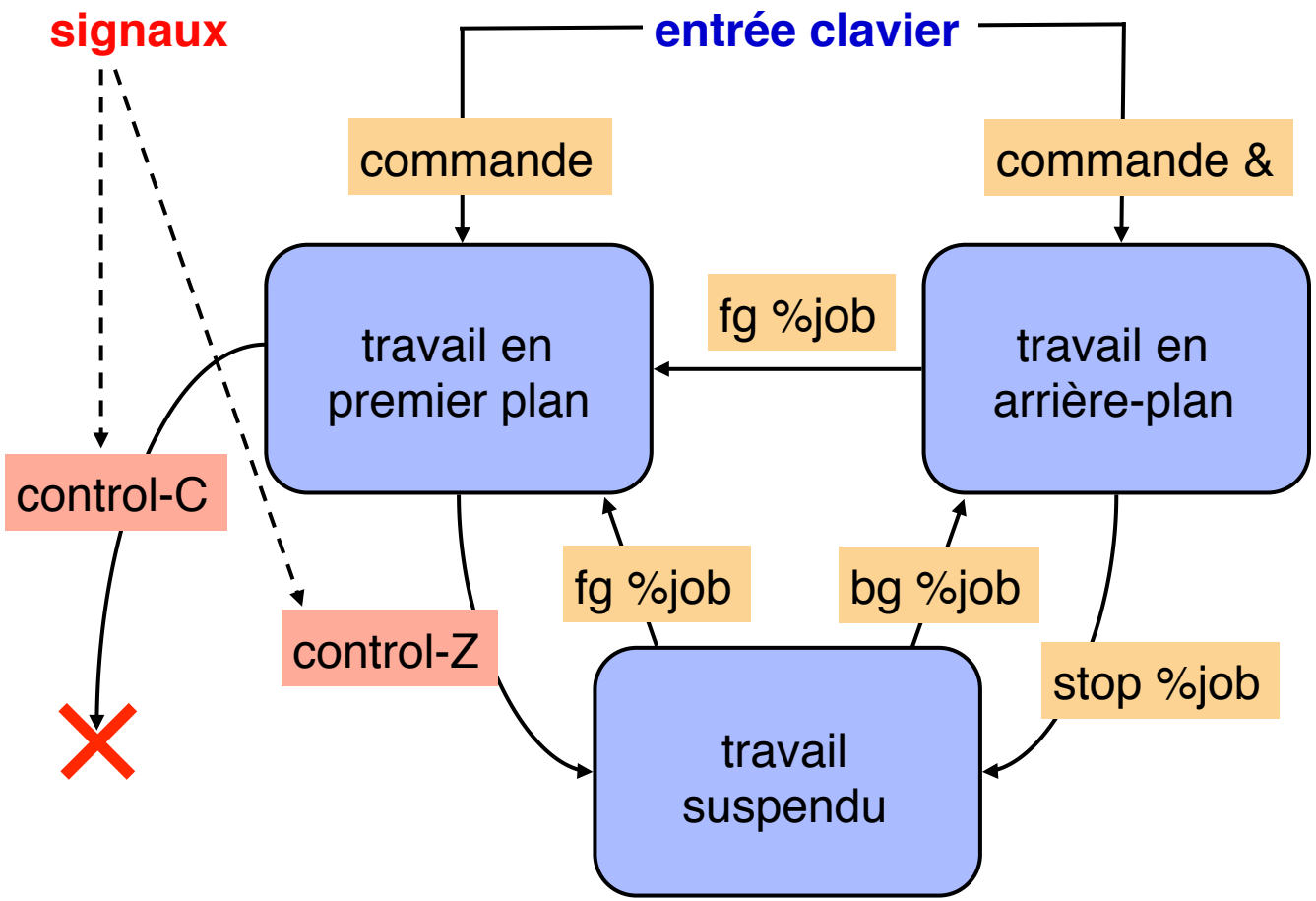
loop.py

```
import os
print "processus %d, groupe %d" %(os.getpid(), os.getpgrp())
while(True)
        pass
```

```
<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   zsh
10468 pts/0   00:00:00   Python
10469 pts/0   00:00:00   Python
10470 pts/0   00:00:00   ps
<unix>fg %1
loop
        =>frappe de control-Z
Suspended
<unix>jobs
[1] + Suspended          Python
[2] - Running            Python
<unix>
```

```
<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   zsh
10468 pts/0   00:02:53   Python
10474 pts/0   00:00:00   ps
<unix>  => frappe de control-C
<unix>ps
  PID TTY          TIME   CMD
 5691 pts/0   00:00:00   zsh
10468 pts/0   00:02:57   Python
10474 pts/0   00:00:00   ps
<unix>
```

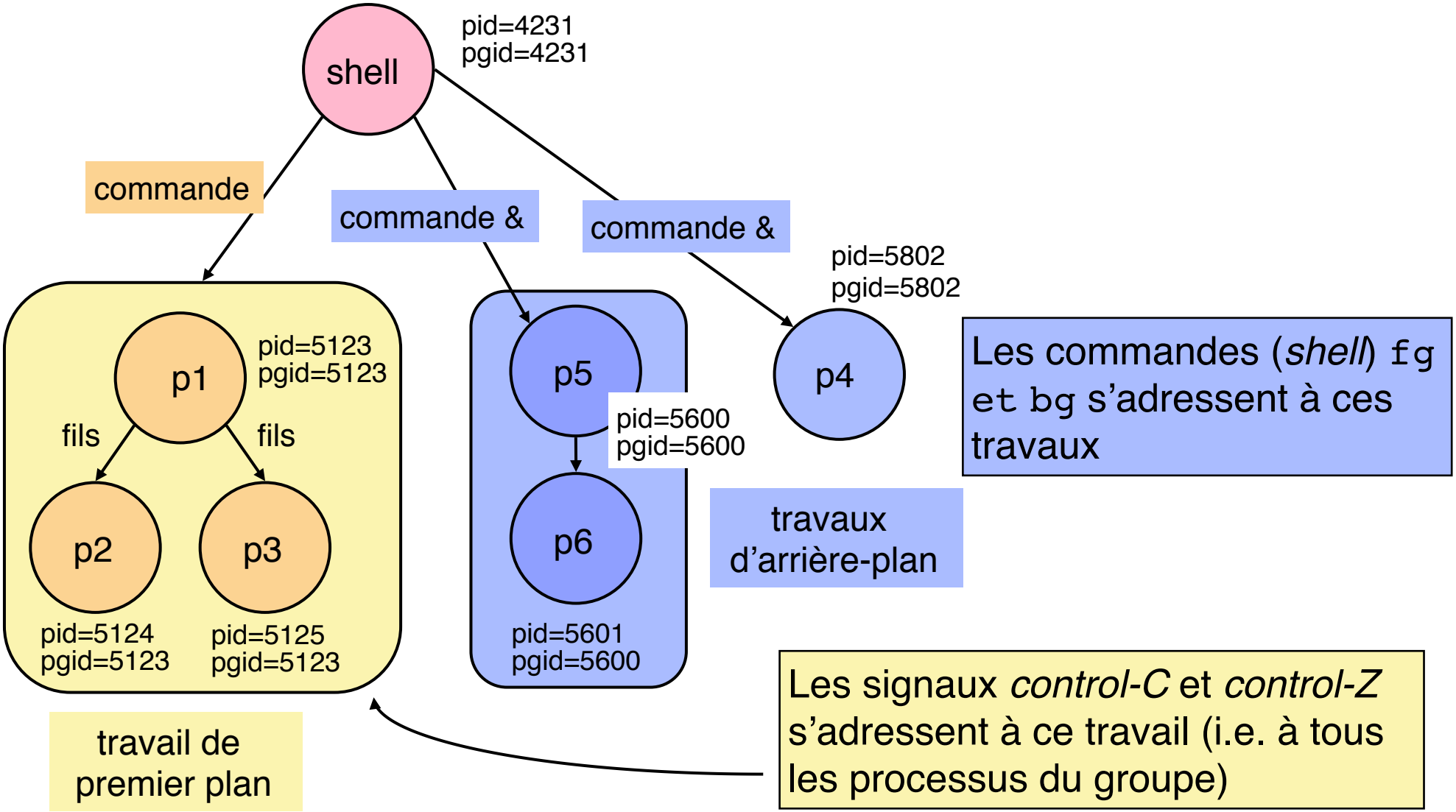
États d'un travail



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



Envoi d'un signal

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

Utilisation : os.`kill(pid, 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 (sauf celui de numéro `-1`)
- 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 de signal (pour des raisons de sécurité) ; en effet il est l'ancêtre de tous les processus, et sa mort entraînerait celle de tous...). Il est donc protégé.

Quelques exemples d'utilisation des signaux

On va donner plusieurs exemples d'utilisation des signaux

- Interaction avec le travail de premier plan : SIGINT, SIGSTOP
- 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

En langage C, on utiliserait 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 python `signal.signal` qui “enveloppe” ces éléments :

```
import signal
def handler(signum,frame):
      <instructions du traitant>

signal.signal(signum, 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.py

```
#! /usr/bin/env python
import signal,sys

def handler(sig,frame):
  print "signal SIGINT reçu !"
  sys.exit(0)

# Programme principal
signal.signal(signal.SIGINT, handler) # installe le traitant
signal.pause()                        # attend un signal
sys.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

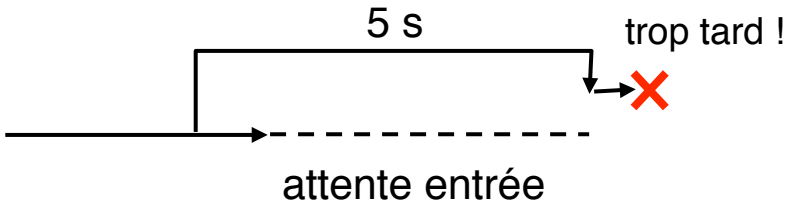
La primitive

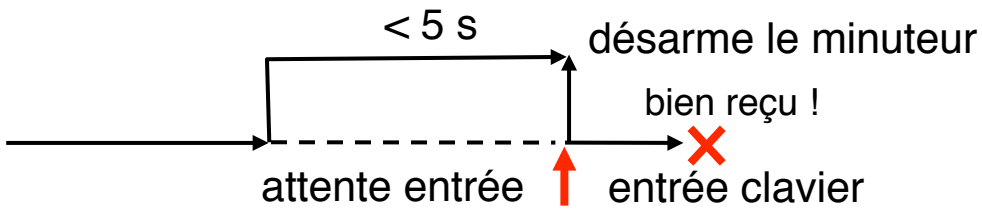
```
import signal
signal.alarm(nbSec)
```

provoque l'envoi du signal SIGALRM après environ `nbSec` secondes ; annulation avec `nbSec` =0

```
import signal,sys
def handler(sig,frame):
  print "trop tard !"
  sys.exit(0)

# Programme principal
signal.signal(signal.SIGALRM, handler) # installe le traitant
signal.alarm(5)
reponse = input("entrer un nombre avant 5 sec. :")
signal.alarm(0)
print "bien reçu !"
sys.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).

```
import signal,sys
def handler(sig,frame):              # nouveau traitant
  (pid,statut) = os.waitpid(-1, 0) # attend un fils quelconque
  return

# Programme principal
signal.signal(signal.SIGCHLD, handler) # installe le traitant
... <création d'un certain nombre de fils, sur demande> ...
sys.exit(0)
```

Autres exemples en TD et TP

Résumé de la séance 2

- **Mise en œuvre des processus**
  - Allocation du processeur : principe, aspects techniques
  - États d'un processus
  - Organisation de la mémoire virtuelle
  - Terminaux, groupes et sessions
- **Communication par signaux**
  - Principe
  - Traitement des signaux
  - Relations entre signaux et interruptions
  - Exemples d'utilisation

Complément pour le projet (mini-shell)

Olivier Dalle
Université de Nice - Sophia Antipolis
http://deptinfo.unice.fr/
D'après le cours original de
Sacha Krakowiak
Université Joseph Fourier
Projet Sardes (INRIA et IMAG-LSR)

http://sardes.inrialpes.fr/~krakowia

Gestion des Groupes de Processus (Berkeley – BSD puis POSIX)

- **Une commande = 1 groupe de processus**
  - Exemple : ls –al | sort –k1n | more
- **Un tty (« ti-ti-ouaille » : terminal)**
  - Associé au groupe d'avant plan
  - Seul autorisé a lire/ecrire sur le tty

tty de Contrôle

- Ouverture possible via /dev/tty)
- Exemple:
  - ps aux : colonne TTY (ou TT)
  - ?? si non associé
  - ttyXX si associé
- Surveille certaines combinaisons de touches
  - Ctrl-Z, Ctrl-C
- Ctrl-Z: tous les processus du groupe recoivent le signal
  - Rendent le contrôle au shell
  - Shell peut faire passer un autre en avant plan

Signaux d'Alerte

- **Ecriture/lecture depuis un processus du groupe d'arrière plan ?**
  - **Opération impossible… Que faire?**
    - **Tuer le processus fautif?**
    - **L'endormir!**
- **SIGTTIN/SIGTTOUT**
  - **Envoyé par le pilote du tty aux processus qui tentent un lecture ou une écriture**
  - **Action par défaut: Stop le processus (envoie SIGSTOP = non maskable)**
    - **De même que Ctrl-Z**

Les primitives (python 3.4)

- os.setpgid(pid, grpid)
  - Deplace un processus (pid) dans un nouveau groupe (grpid)
    - Nouveau groupe: grpid = pid
    - Pid = 0: processus courant
  - os.setpgrp() : idem os.setpgid(0,0)
- int getpgrp(int pid)
  - Retrouve legroupe du processus pid
    - Pid = 0 : processus courant
- os.killpg(grpid,sig)
  - Envoie le signal sig a tous les processus du groupe
  - Idem os.kill(-pgid,sig)

Primitive de gestion du terminal

- **os.tcsetpgrp(fd,pgid)**
  - Assigne le groupe pgid au terminal associé au descripteur fd
  - Une version simplifiée de la très compliquée fcntl.ioctl…
  - Attention: ne fonctionne que si le tty est deja le « controling tty »
- **os.tcgetpgrp(fd)**
  - Récupère le groupe (pgid) associé à un tty
- **os.ttyname(fd)**
  - Une chaine qui specifie le peripherique tty
  - Exception si non associé a un tty

Création d'une session

- **os.setsid()**
  - S'applique au precessus courant:
    - Devient leader de session
    - Devient leader de groupe
    - Sans terminal de contrôle
    - Seul dans sa session et son groupe
  - Limitations:
    - echoue si deja leader de groupe
    - Un autre processus a son pgid = pid de l'appelant
  - Utilisé pour créer des processus « demons »
    - Tournent en permanence => impossible de les atatcher a un terminal
    - Ecrivent dans les logs ou la console systeme (/dev/console)