Processus (suite)

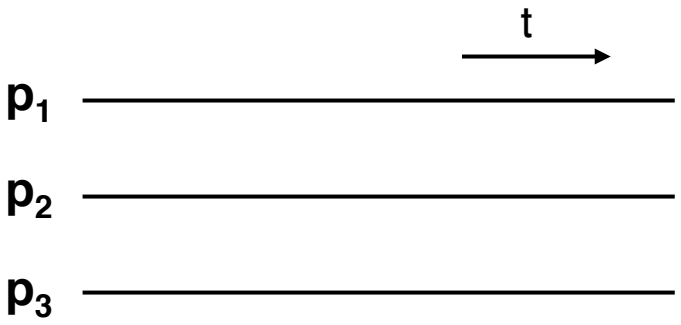
Réalisation des processus
Communication par signaux

Noel De Palma

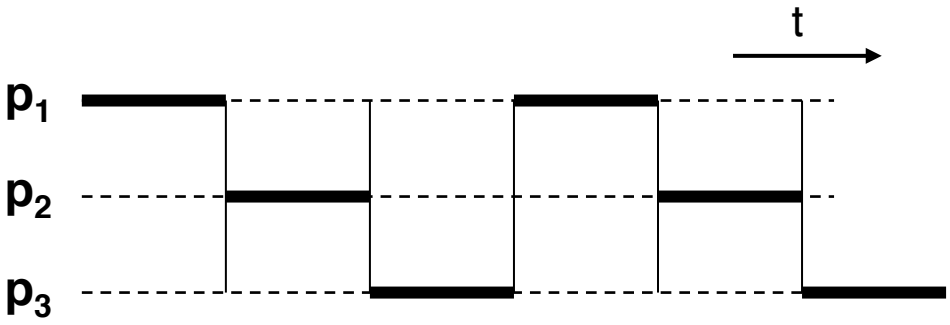
Ce cours est basé sur les transparents de Sacha Krakowiak/Renaud Lachaize

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)

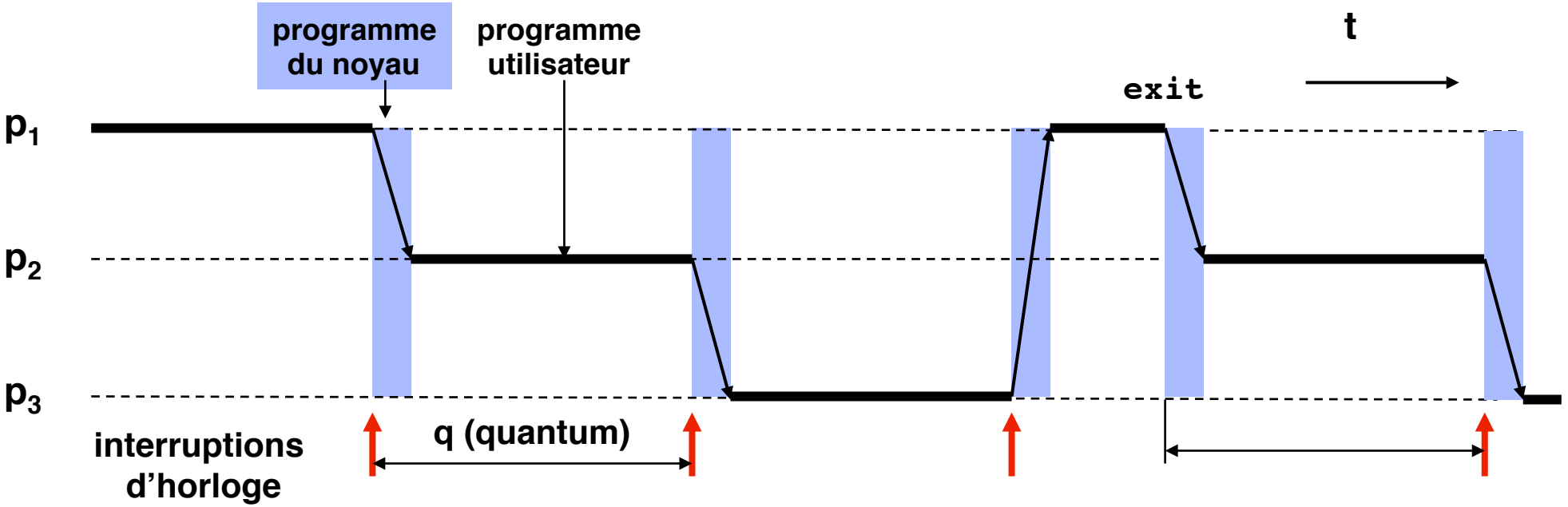


vue abstraite
t = temps logique



vue concrète (très simplifiée)
t = temps physique

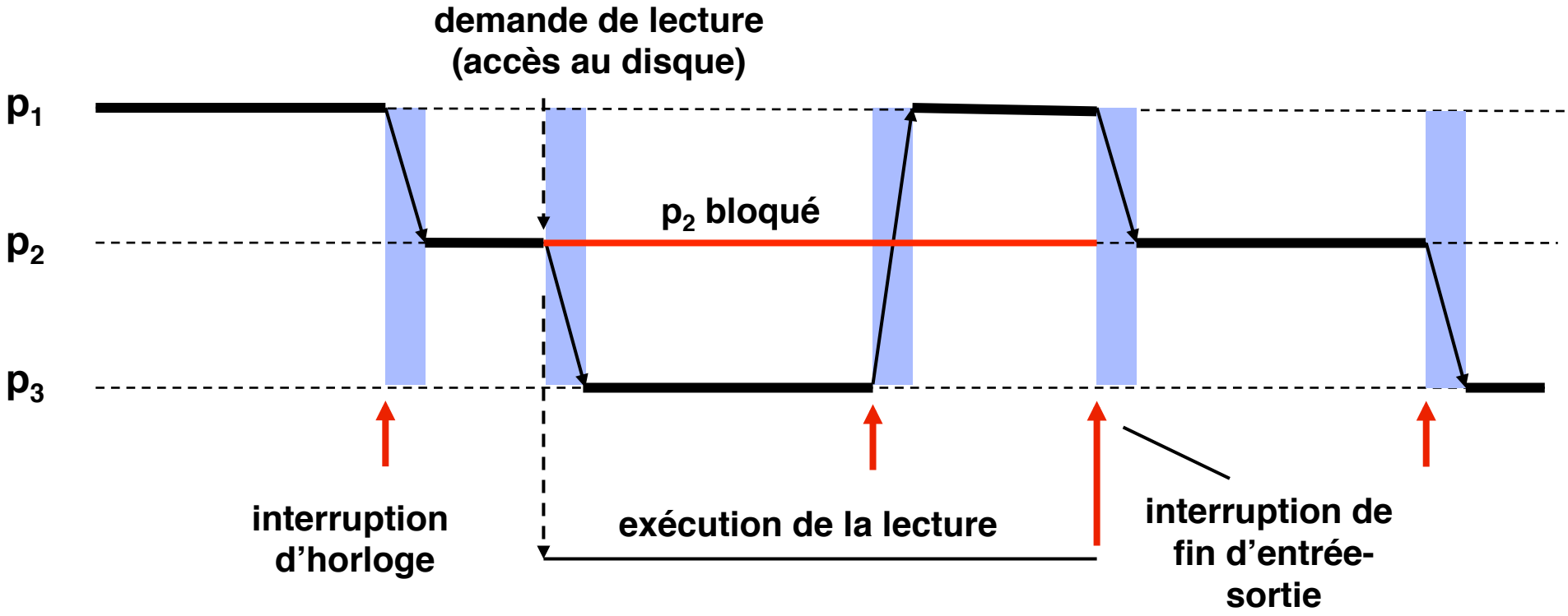
Allocation du processeur aux processus (1)



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*)

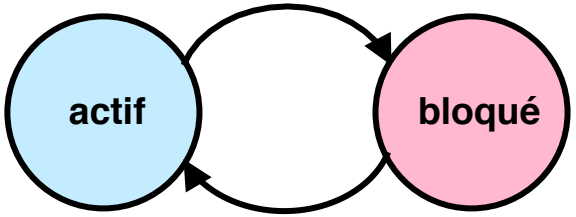
Allocation du processeur aux processus (2)



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

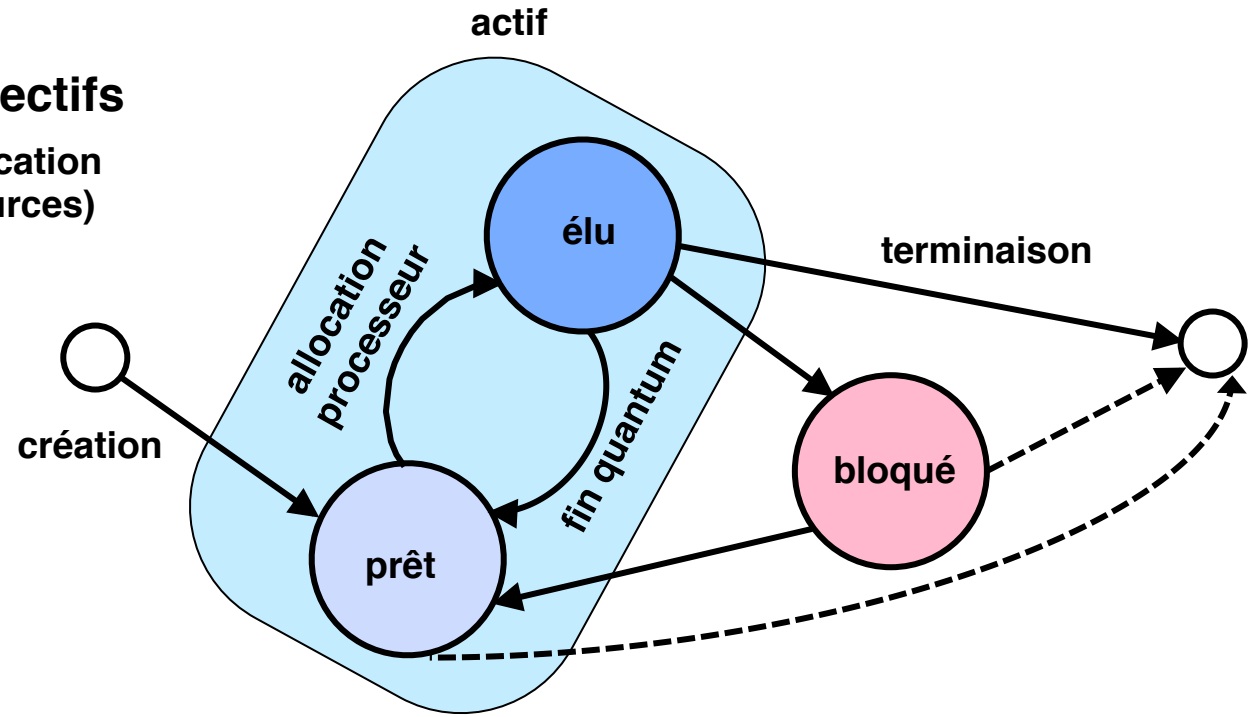


causes de blocage :

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

Communication entre processus dans un système 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 un système 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 ultérieur cette année]
- Communication par **files de messages** [non étudiée, voir cours de M1]
- Communication par **mémoire partagée** (brièvement évoquée cette année, détails en M1) et par **sémaphores** [non étudiée, voir cours de M1]
- Communication par ***sockets***, utilisés dans les réseaux, mais aussi en local [à voir dans cours ultérieur cette année]

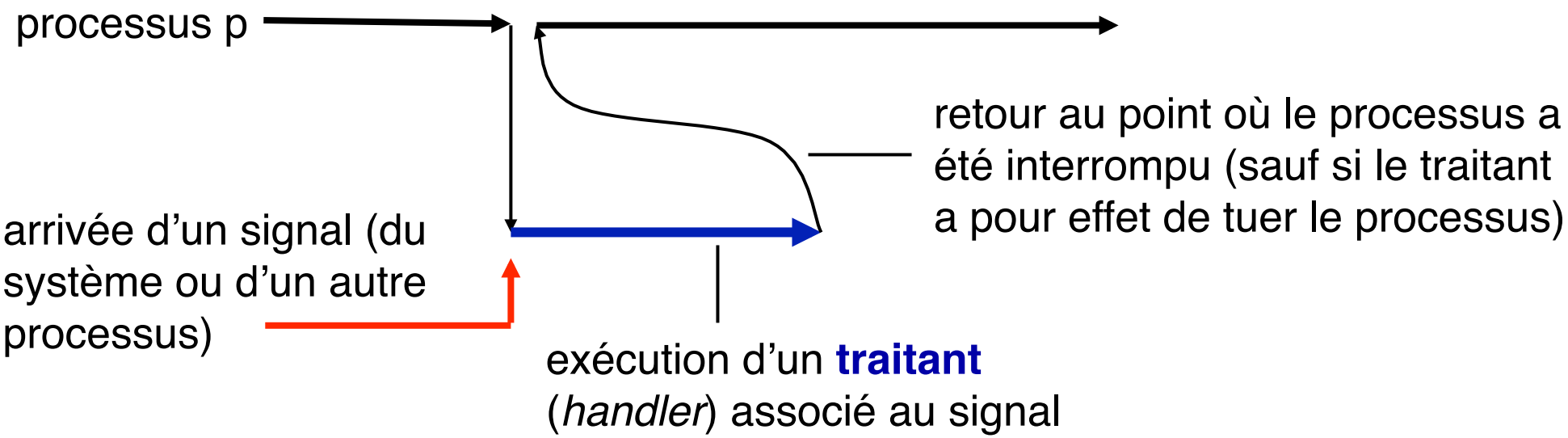
Signaux

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** (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
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**

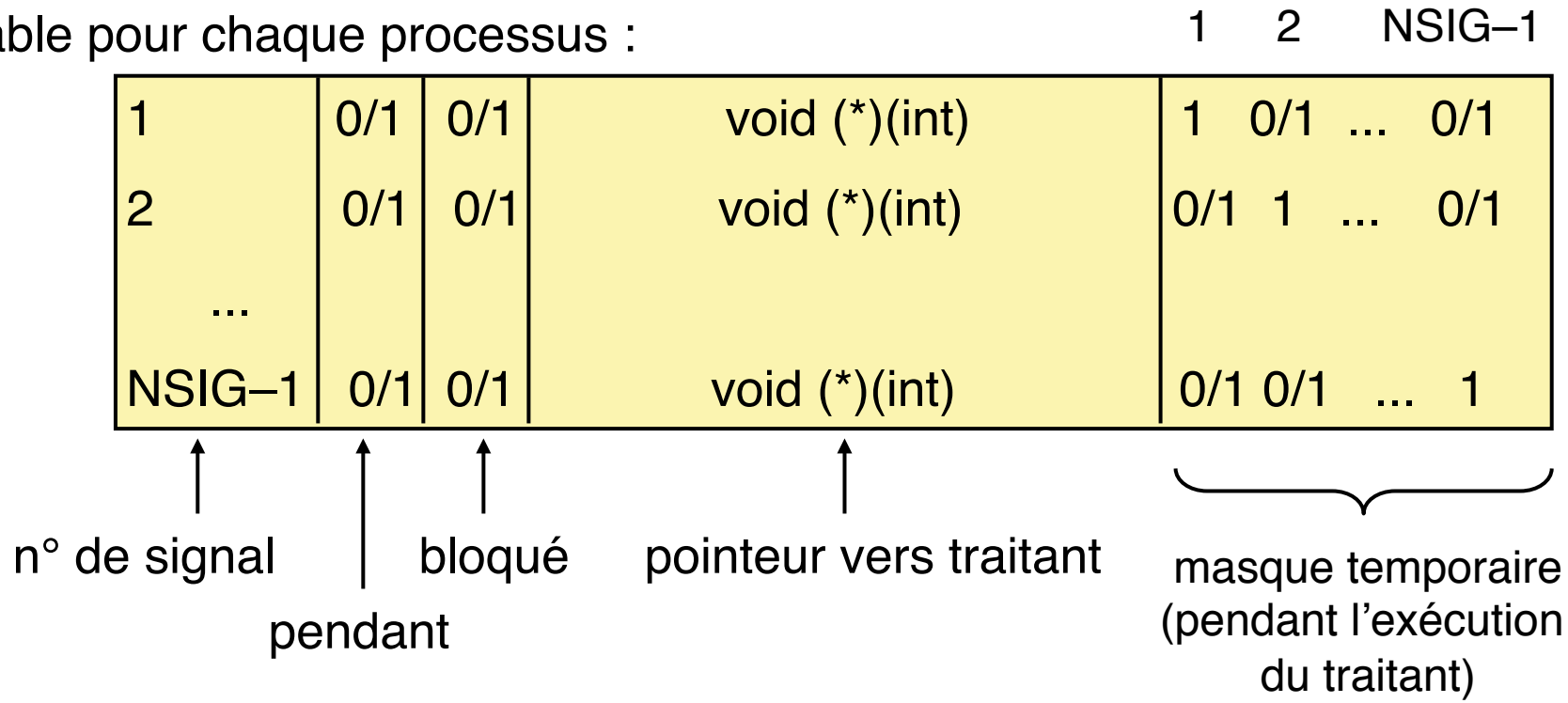
Qu'est-ce qui empêche que le signal soit immédiatement traité dès qu'il est reçu ?

- 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

Structures internes associées aux signaux

Table pour chaque processus :



Quand un signal d'un type i donné est reçu, pendant[i]=1. Alors si 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.

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)

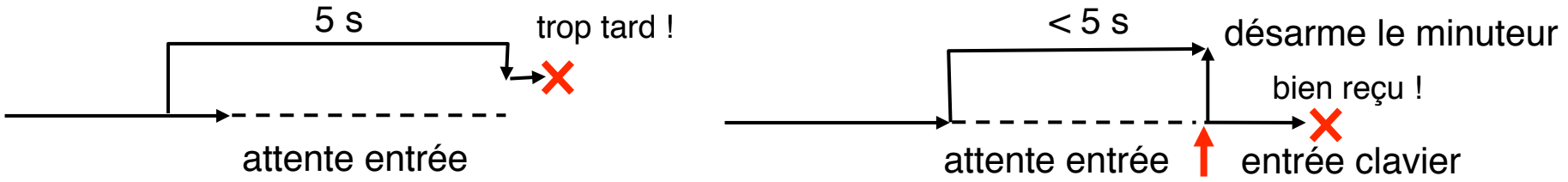
Exemple 2 : utilisation de la temporisation

La primitive

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

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

```
#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),
}
```

Autres exemples en TD et TP

Gestion du masque de signaux

Le masque de signaux d'un processus est automatiquement modifié lors de l'exécution d'un traitant (en fonction des consignes spécifiées au moment de la définition des traitants). Il y a blocage/masquage d'un ou plusieurs types de signaux (au moins le signal en cours de traitement). On ne modifie pas explicitement le masque de signaux dans le code d'un traitant.

En revanche, on peut modifier (et consulter) le masque de signaux d'un processus dans le code extérieur aux traitants avec la primitive `sigprocmask` :

```
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
  ...
}
```

Voir autres exemples en TD et TP (et `man 2 sigprocmask`)

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. 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 sous Unix (2)

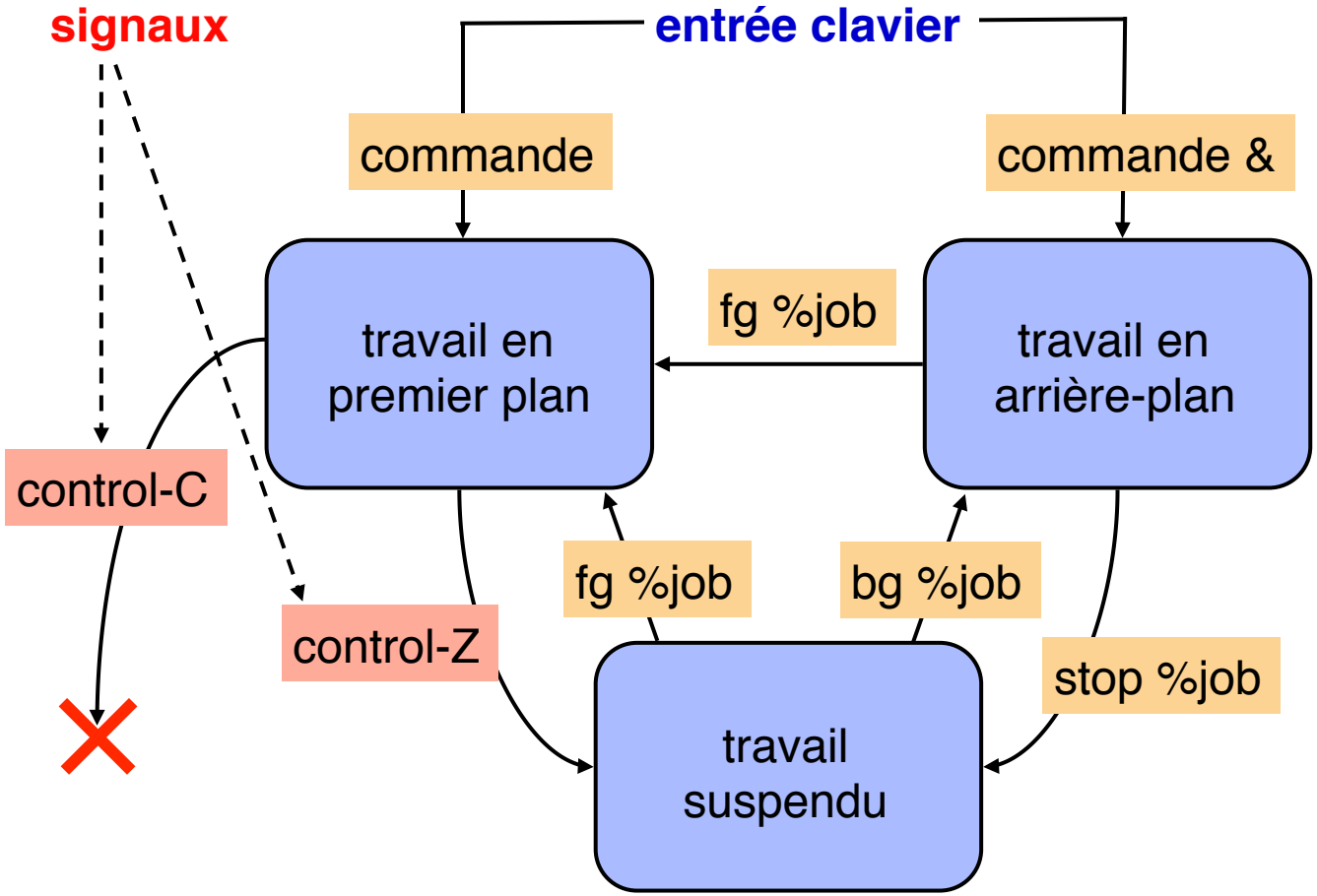
loop.c

```
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>
```

```
<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>
```

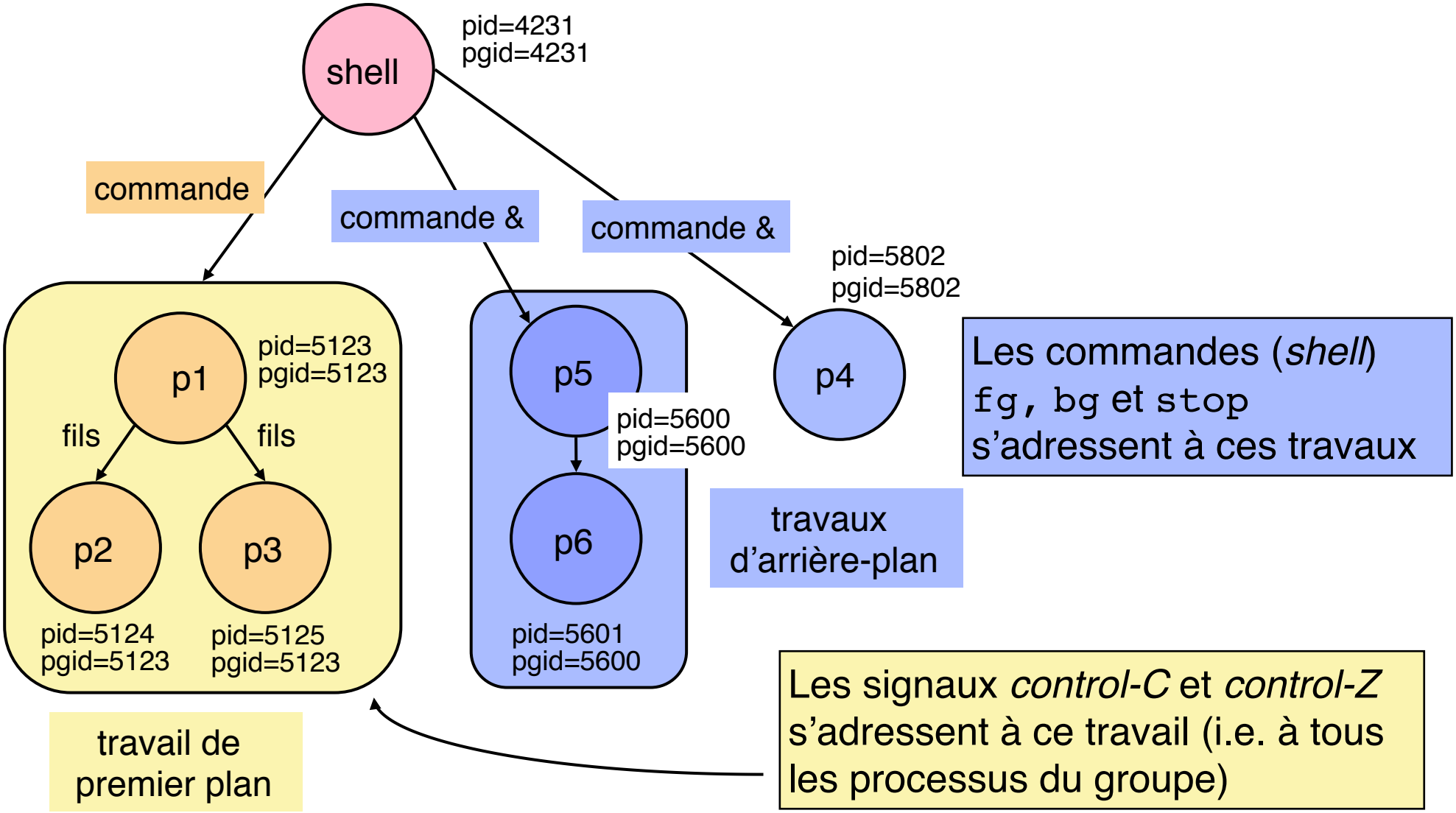
États d'un travail



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



Résumé de la séance 2

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