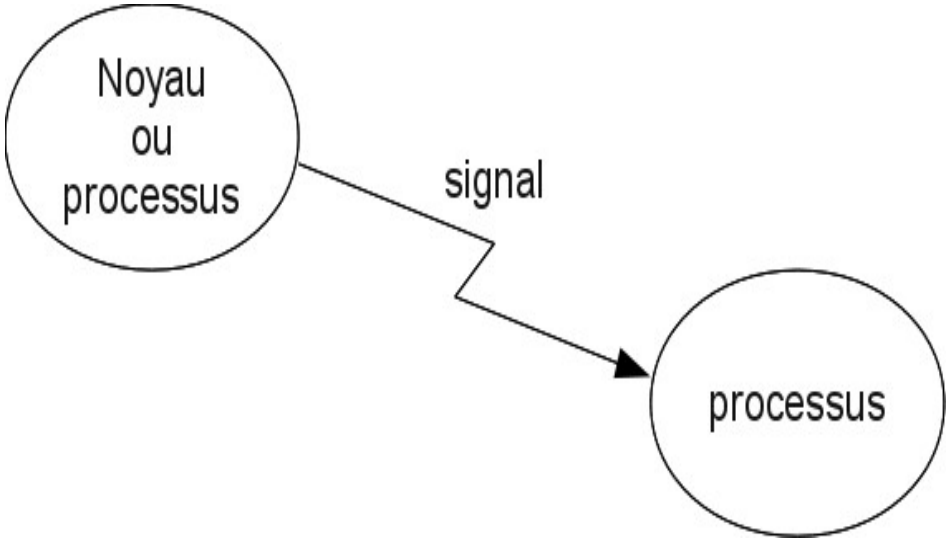
# Programmation Système : les signaux



Un mécanisme de synchronisation de processus : les **signaux** (ou les **évènements**) permettent de communiquer entre processus : réveiller, arrêter ou avertir un processus d'un événement.




Un signal peut être imaginé comme une sorte d'impulsion qui oblige le processus cible à prendre immédiatement une mesure spécifique.

L'utilisation des signaux sous Linux est décrite dans : **`man 7 signal`**

# La communication inter-processus

- Multi-tâches :
  Les processus ne sont pas isolés dans l'ordinateur puisqu'ils concourent tous à l'exploitation de travaux de l'utilisateur et puisqu'ils sont en compétition pour le partage de ressources qui sont limitées.

  ⇥ Ceci implique un besoin des processus de communiquer.

- Multi-programmation :
  L'idée (paralléliser des traitements) est de substituer un seul programme purement séquentiel en plusieurs pièces (processus ou tâches) pouvant logiquement se dérouler simultanément et en partageant des données.

  ⇥ Ceci implique un besoin des processus de communiquer.

# Les communications inter-processus (IPC)

- Les communications inter processus (*Inter-Process Communication* ou IPC) regroupent un ensemble de mécanismes permettant à des processus concurrents (ou distants) de communiquer.

- Ces mécanismes peuvent être classés en trois catégories :

  - les outils permettant aux processus de s'échanger des données ;

  - les outils permettant de synchroniser les processus, notamment pour gérer le principe de section critique ;

  - les outils offrant directement les caractéristiques des deux premiers (échanger des données et synchroniser des processus).

# Les signaux

- Le rôle des signaux est de permettre aux processus de communiquer. Ils peuvent par exemple :
  - réveiller un processus
  - arrêter un processus
  - avertir un processus d'un événement

- À la réception d'un signal, le processus peut :
  - l'ignorer
  - déclencher l'action prévue par défaut sur le système
  - déclencher un traitement spécial appelé *handler* (gestionnaire du signal)

# Les signaux disponibles

- Sur un système donné, on dispose de NSIG signaux numérotés de 1 à NSIG. Les différents signaux et les prototypes des fonctions qui les manipulent sont définis dans le fichier **<signal.h>**.

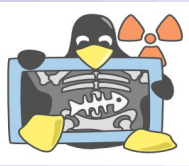
```
$ grep NSIG /usr/include/bits/signum.h
#define _NSIG           65      /* Biggest signal number + 1

$ grep SIGKILL /usr/include/bits/signum.h
#define SIGKILL         9       /* Kill, unblockable (POSIX).  */
```

- La commande **kill**, qui permet d'envoyer un signal à un processus, nous donne aussi la liste des signaux :

```
$ kill -l
 1) SIGHUP    2) SIGINT    3) SIGQUIT   4) SIGILL    5) SIGTRAP
 6) SIGABRT   7) SIGBUS    8) SIGFPE    9) SIGKILL   10) SIGUSR1
 etc ...
```

# Terminologie



- Un signal envoyé par le noyau ou par un autre processus est un signal **pendant** : cet envoi est mémorisé dans le BCP du processus.

- Un signal est **délivré** (ou pris en compte) lorsque le processus concerné réalise l'action qui lui est associée dans son BCP, c'est à dire, au choix :

  - l'action par défaut (souvent la mort du processus)

  - ignorer le signal

  - l'action définie par l'utilisateur (handler) : le signal est dit **capté**.

- Un signal peut également être **masqué** (ou **bloqué**) : sa prise en compte sera différée jusqu'à ce que le signal ne soit plus masqué.

# L'envoi des signaux

- La commande **`kill`** permet d'envoyer un signal à un processus ou à un groupe de processus :

```
$ kill [-s signal] pid... # man kill pour plus de détails
```

*Par exemple, pour envoyer le signal SIGINT au processus de PID 2000 :*

```
$ kill -2 2000   ou   $ kill -INT 2000
```

- L'appel système **`kill()`** peut être utilisé pour envoyer n'importe quel signal à n'importe quel processus ou groupe de processus :

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

***Exemple d'utilisation :***

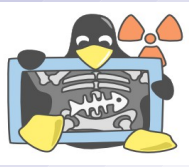
```
int interrompre(int pid) {
   if (kill(pid, SIGINT) == 0) return 0; // ou : kill(pid, 2);
   else perror("kill");
   return -1;
}
```

*Remarque : La primitive raise permet à un processus de s'envoyer un signal.*

# Interception d'un signal

- La primitive **`signal()`** permet d'associer un traitement à la réception d'un signal d'interruption. Une autre primitive, **`sigaction()`**, permet de faire la même chose et présente l'avantage de définir précisément le comportement désiré et de ne pas poser de problème de compatibilité.
- **`signal()`** installe le gestionnaire *handler* pour le signal *signum*. *handler* peut être SIG_IGN, SIG_DFL ou l'adresse d'une fonction définie par le programmeur (un « gestionnaire de signal »).
- Lors de l'arrivée d'un signal correspondant au numéro **signum**, un des événements suivants se produit :
  - Si le gestionnaire est SIG_IGN, le signal est ignoré.
  - Si le gestionnaire est SIG_DFL, l'action par défaut associé à ce signal est entreprise.
  - Si le gestionnaire est une fonction, alors *handler* est appelée avec l'argument *signum*.

# Attente d'un signal



- L'attente d'un signal démontre tout l'intérêt du multi-tâche. En effet, au lieu de consommer des ressources CPU inutilement en testant la présence d'un événement dans une boucle `while`, on place le processus en sommeil et le processeur est mis alors à la disposition d'autres tâches.

- La primitive **`pause()`** endort le processus appelant et le place dans l'attente d'un signal d'interruption (quelconque). Le problème qui se pose est d'encadrer correctement l'appel `pause`. Il faudra aussi regarder du côté des appels **`sigsuspend()`** et **`sigwait()`**.


**tp-sys-signaux.pdf**


Séquence 1 : les bases

# Gestion des délais

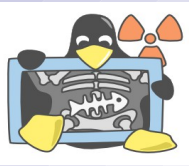
- La primitive **`alarm()`** initie un délai en secondes à l'expiration duquel le signal **SIGALRM** sera reçu.

- La temporisation peut être désarmée en passant en paramètre la valeur 0 à `alarm`. Dans ce cas, la fonction renvoie comme résultat le nombre de seconde qu'il restait avant expiration du délai.


**tp-sys-signaux.pdf**

Séquence 0 : exemples

# Temporisation



- Trois autres fonctions permettent d'endormir un processus : **`sleep`**, **`usleep`**, **`nanosleep`**.

- *Remarque : La période de sommeil peut être allongée par la charge système, par le temps passé à traiter l'appel de fonction, ou par la granularité des temporisations système. La précision de toutes ces fonctions est donc toute relative et ne dépasse pas dans le meilleur des cas **10 ms**. Pour obtenir une meilleure précision la fonction **`setitimer`** doit être utilisée.*


**tp-sys-signaux.pdf**


Séquence 2 : gestion des délais

# Blocage des signaux

- Un processus peut bloquer à volonté un ensemble de signaux, à l'exception de SIGKILL et de SIGSTOP.

- Cette opération se fait par l'appel système **`sigprocmask()`**. Cette fonction permet aussi bien de bloquer ou de débloquer des signaux, que de fixer un nouveau masque ou de consulter l'ancien masque de blocage.

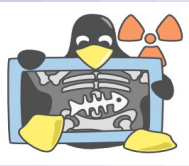
- L'utilité principale d'un blocage des signaux est la protection des portions critiques de code. Il est à noter que la délivrance des signaux s'effectue avant le retour de la fonction `sigprocmask`.

- Un processus peut consulter la liste des signaux bloqués sans en demander la délivrance immédiate avec **`sigpending()`**.

# Limites

- Lorsqu'un processus est en sommeil et qu'il reçoit plusieurs signaux :
  - Aucune mémorisation du nombre de signaux reçus :
    10 signaux SIGINT = 1 seul signal SIGINT.
  - Aucune mémorisation de la date de réception d'un signal :
    les signaux seront traités ultérieurement par ordre de numéro.
  - Aucun moyen de connaître le PID du processus émetteur du signal
    (sauf si on utilise siginfo_t).
- Attention, les signaux sont **asynchrones** : la délivrance des signaux non masqués a lieu un « certain temps » après leur envoi, quand le processus récepteur passe de l'état actif noyau à l'état actif utilisateur. Cela explique pourquoi un processus n'est pas interruptible lorsqu'il exécute un appel système (dans le noyau).

# Conclusion



- Les signaux assurent la notification d'évènements entre processus ou entre le noyau et des processus. C'est un moyen de communication entre processus afin de leurs permettre un échange ou une synchronisation.
- Il est conseillé avant de continuer de revoir :
  - Les principes et la terminologie associée aux signaux
  - L'exemple fourni pour l'environnement Windows (event)
- Il reste à voir entre autres :
  ordonnancement des processus,
  autres types de communications entre processus


**tp-sys-signaux.pdf**

Bonus : essuie-glace