Sommaire

Séquence 0 : exemples

Interception d'un signal

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 »).

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

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.

Remarques : Les signaux SIGKILL et SIGSTOP ne peuvent être ni ignorés, ni interceptés.

Il existe de nombreux systèmes Unix sur lesquels un gestionnaire de signal ne reste pas en place après avoir été invoqué. Le noyau restaure alors le comportement par défaut. Donc, par souci de compatibilité, il est prudent de systématiquement réinstaller le gestionnaire de signal au moment de son appel. Il est toutefois possible que le signal arrive de nouveau avant que le gestionnaire ne soit réinstallé.

Exemple d'utilisation (compteur1.c) :

```
int compteur; /* variable globale */

void gestionnaire(int numSig)
{
  signal(numSig, gestionnaire); /*réinstaller le gestionnaire de
signal*/
  printf("Compteur=%d\n", compteur++);
}

main()
{
  printf("Je suis le PID=%d\n", getpid());
  signal(SIGUSR1, gestionnaire); /* installe un gestionnaire de signal
pour SIGUSR1 */
  while(1); /* peut faire autre chose ... */
}
```

Ce qui donne :

```
$ ./compteur
Je suis le PID=12933
Compteur=1
Compteur=2
^C
```

Et dans une autre console :

```
$ kill -USR1 12933
$ kill -USR1 12933
```

L'autre façon de mettre en place un gestionnaire de signal est d'utiliser la primitive **sigaction()** :

```
int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);
```

Le deuxième paramètre correspond au nouveau comportement à adopter et le troisième paramètre correspond à l'ancien comportement sauvegardé. Le comportement à adopter est décrit par une structure :

```
struct sigaction
{
  __sighandler_t sa_handler; /* gestionnaire de signal */
  unsigned long  sa_flags;   /* comportement du signal */
  void           (*sa_restorer)(void); /* obsolète, ne pas utiliser */
  sigset_t       sa_mask;    /* masque des signaux à bloquer */
};
```

Pour plus de détails, faire : **`man sigaction`**

Exemple d'utilisation (compteur2.c) :

```
int compteur = 1; /* variable globale */

void gestionnaire(int numSig)
{
   printf("Compteur=%d\n", compteur++);
}

main()
{
   struct sigaction action;

   printf("Je suis le PID=%d\n", getpid());

   action.sa_handler = gestionnaire;
   sigemptyset(&action.sa_mask);
   action.sa_flags = 0;
   sigaction(SIGUSR1, &action, NULL);

   while(1); /* peut faire autre chose ... */
}
```

On peut aussi utiliser un autre type de gestionnaire (sa_sigaction) qui permettra notamment de connaître le PID du processus émetteur du signal.

Exemple d'utilisation (compteur3.c) :

```
int compteur = 1; /* variable globale */

void gestionnaire(int numSig, siginfo_t *siginfo, void *context)
{
   printf("Le processus %d m'a envoyé le signal %d (%s)\n", siginfo-
>si_pid, siginfo->si_signo, sys_siglist[numSig]);
   printf("Compteur=%d\n", compteur++);
}

main()
{
   struct sigaction action;
   printf("Je suis le PID=%d\n", getpid());
   action.sa_sigaction = gestionnaire;
   sigemptyset(&action.sa_mask);
   action.sa_flags = SA_SIGINFO;
   sigaction(SIGUSR1, &action, NULL);

   while(1); /* peut faire autre chose ... */
}
```

Ce qui donne :

```
$ ./compteur3
Je suis le PID=18767
Le processus 3167 m'a envoyé le signal 10 (User defined signal 1)
Compteur=1
Le processus 3167 m'a envoyé le signal 10 (User defined signal 1)
Compteur=2
Le processus 18963 m'a envoyé le signal 10 (User defined signal 1)
Compteur=3
^C
```

Blocage d'un signal

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.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

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 :

```
int sigpending(sigset_t *set) ;
```

Exemple d'utilisation (pending.c) :

```
void gestionnaire(int numSig)
{
  printf("Signal %d (%s) reçu\n", numSig, sys_siglist[numSig]);
}

int main()
{
   int i;
   struct sigaction action;
   sigset_t ensemble;

   printf("Je suis le PID=%d\n", getpid());

   /* on installe le gestionnaire */
   action.sa_handler = gestionnaire;
   sigemptyset(&action.sa_mask);
   action.sa_flags = 0;

   /* on intercepte tous les signaux */
   for (i=1;i<NSIG;i++) sigaction(i, &action, NULL);

   /* on bloque tous les signaux sauf SIGINT */
   sigfillset(&ensemble);
   sigdelset(&ensemble, SIGINT);
   sigprocmask(SIG_BLOCK, &ensemble, NULL);

   /* appel système lent */
   read(0, &i, sizeof(int)); /* on continue par une saisie */

   /* quels sont les signaux en attente ? */
   sigpending(&ensemble);
   for (i=1;i<NSIG; i++)
      if (sigismember(&ensemble, i))
         printf("Signal %d (%s) en attente\n", i, (i>31) ? "sans
nom" : sys_siglist[i]);

   /* on débloque tous les signaux */
   sigemptyset(&ensemble);
   sigprocmask(SIG_SETMASK, &ensemble, NULL);

   return 0;
}
```

Ce qui donne :

```
$ ./pending
Je suis le PID=21787
1
Signal 1 (Hangup) en attente
Signal 10 (User defined signal 1) en attente
Signal 12 (User defined signal 2) en attente
Signal 12 (User defined signal 2) reçu
Signal 10 (User defined signal 1) reçu
Signal 1 (Hangup) reçu
```

Les signaux envoyés au processus 21787 pendant l'appel read() :

```
$ kill -USR1 21787
$ kill -USR2 21787
$ kill -HUP 21787
$ kill -USR1 21787
```

Remarques :

Les signaux classiques ne sont pas empilés et ne sont pas délivrés selon leur ordre chronologique d'occurrence. Il n'existe pas non plus de priorité entre les signaux.

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).

```
int pause(void);
```

Le problème qui se pose est d'encadrer correctement pause. Des exemples sont fournis sur le serveur.

Il existe un appel système **sigsuspend()**, qui permet de manière atomique de modifier le masque des signaux et de bloquer en attente. Une fois qu'un signal arrive, sigsuspend restitue le masque d'origine avant de se terminer. L'ensemble transmis est celui des signaux qu'on bloque et non pas celui des signaux qu'on attend.

```
int sigsuspend(const sigset_t *mask);
```

Il existe encore la fonction **sigwait()** qui suspend l'exécution jusqu'à l'interception de l'un des signaux indiqués dans l'ensemble de signaux set. La fonction accepte le signal (le supprime de la liste des signaux en attente), et retourne le numéro du signal dans sig.

```
int sigwait(const sigset_t *set, int *sig);
```

Gestion des délais

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

```
unsigned int alarm(unsigned int nb_sec) ;
```

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.

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

```
unsigned int sleep(unsigned int nb_sec) ;

int usleep(useconds_t usec) ;

int nanosleep(const struct timespec *req, struct timespec *rem);

struct timespec {
  long  tv_sec;  // nombre de secondes
  long  tv_nsec; // nomre de nanosecondes
};
```

sleep() peut être implémenté en utilisant SIGALRM. Ainsi l'utilisation conjointe de alarm et sleep est une très mauvaise idée. Un processus endormi par sleep peut être réveillé par l'arrivée d'un signal. Dans ce cas, sleep renvoie le nombre de secondes qu'il restait. nanosleep renvoie comme valeur 0 si le sommeil est allé à son terme et –1 sinon.

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.

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.

Pour installer un *handler,* le processus dispose des appels signal() et sigaction() :

```
sighandler_t signal(int signum, sighandler_t handler);
int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);
```

Pour positionner les masques, le processus dispose des fonctions suivantes :

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

L'appel sigpending() permet d'examiner les signaux en attente :

```
int sigpending(sigset_t *set);
```

Et pour se mettre en attente d'un signal, on utilisera pause(), sigsuspend() ou sigwait() :

```
int pause(void);
int sigsuspend(const sigset_t *mask);
int sigwait(const sigset_t *set, int *sig);
```

Lorsqu'un processus décide d'envoyer un signal à un autre processus, on utilisera :

```
int kill(pid_t pid, int sig);
int sigqueue(pid_t pid, int sig, const union sigval valeur);
```

Remarque : Beaucoup de ces appels sont redondants, il est du ressort du programmeur de n'utiliser que ceux dont il a besoin.

En théorie, la seule chose qu'on puisse faire dans un gestionnaire de signal est de modifier une ou plusieurs variables globales de type **sig_atomic_t**. La variable globale doit être déclarée **volatile**.

Il faut par ailleurs s'assurer de la réentrance des fonctions appelées à l'intérieur d'un gestionnaire de signal.

Pour en savoir plus

L'utilisation des signaux sous Linux est décrite dans :

```
man 7 signal
```

Séquence 1 : les bases

Soit le programme `signaux1.c` :

```
#include <stdio.h>
#include <signal.h>

void monTraitement(int numSignal)
{
   // réarmement :
   signal(numSignal, monTraitement);
   printf("J'ai recu le signal %s\n", sys_siglist[numSignal]);
}

int main()
{
   setbuf(stdout, NULL);

   if (signal(SIGUSR1, monTraitement) == SIG_ERR) printf("Signal non
intercepte\n");

   printf("Merci de m'envoyer 3 signaux par la commande :\n");
   printf("kill -USR1 %d \n", getpid(), getpid());
   pause();
   printf("plus que 2 signaux USR1 !\n");
   pause();
   printf("plus que 1 signal USR1 !\n");
   pause();
   printf("C'est termine !!!\n");

   return 0;
}
```

Remarque : Il est important de privilégier l'utilisation du nom symbolique d'un signal et non son numéro, car celui-ci peut varier d'un système à l'autre.

1) Compilez et exécutez ce programme. A quelle condition ce programme peut-il arriver à son terme ?

2) Que se passe-t-il si le 2ème signal envoyé est USR2 ? INT ? Tester.

3) Que se passe-t-il si on ajoute la ligne : signal(2, SIG_IGN); ? Tester avec Ctrl-C.

4) Modifiez le programme pour qu'il accepte indifféremment les signaux USR1 et USR2 lors de son exécution ?

5) Que fait le programme menage.c suivant ?

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void menage(int numSignal)
{
  printf("J'ai recu le signal %s\n", sys_siglist[numSignal]);
  printf("Je fais le menage...\n");
  printf("et je (\"%d\") termine\n", getpid());
  exit(numSignal);
}

int main()
{
  char fichier[255];
  setbuf(stdout, NULL);

  if(signal(SIGUSR1, menage) == SIG_ERR)
    printf("Signal non intercepte\n");
  if(signal(SIGUSR2, menage) == SIG_ERR)
    printf("Signal non intercepte\n");
  if(signal(SIGINT, menage) == SIG_ERR)
    printf("Signal non intercepte\n");
  if(signal(SIGALRM, menage) == SIG_ERR)
    printf("Signal non intercepte\n");

  while (1)
  {
    printf("Entrez un nom de fichier :");
    scanf("%s", fichier);
    printf("et je (\"%d\") traite longuement votre fichier %s\n",
getpid(), fichier);

    sleep(20); // Simulation du traitement du fichier
  }
  return 0;
}
```

Séquence 2 : gestion des délais

On veut produire un message "Bip" périodiquement, toutes les secondes. Pour cela un processus se place dans une boucle infinie, mais arme au préalable une temporisation d'une seconde (cf. alarm). Sur réception du signal SIGALRM, le processus affiche le message et réarme à nouveau la temporisation.

6) Écrire le programme bippeur.c

7) Modifiez le programme pour qu'à la réception du signal :

- SIGUSR1 : la fréquence d'affichage diminue d'une seconde
- SIGUSR2 : la fréquence d'affichage augmente d'un seconde

Remarque : la fréquence d'affichage ne sera jamais inférieure à une seconde.

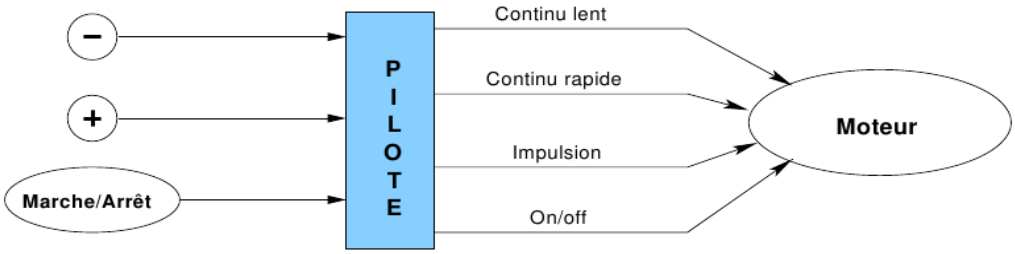
8) Donner (et tester) les commandes kill pour utiliser les signaux adéquats pour :

- suspendre le processus (STOP)
- reprendre son exécution (CONT)
- le terminer (TERM)

Bonus : essuie-glace

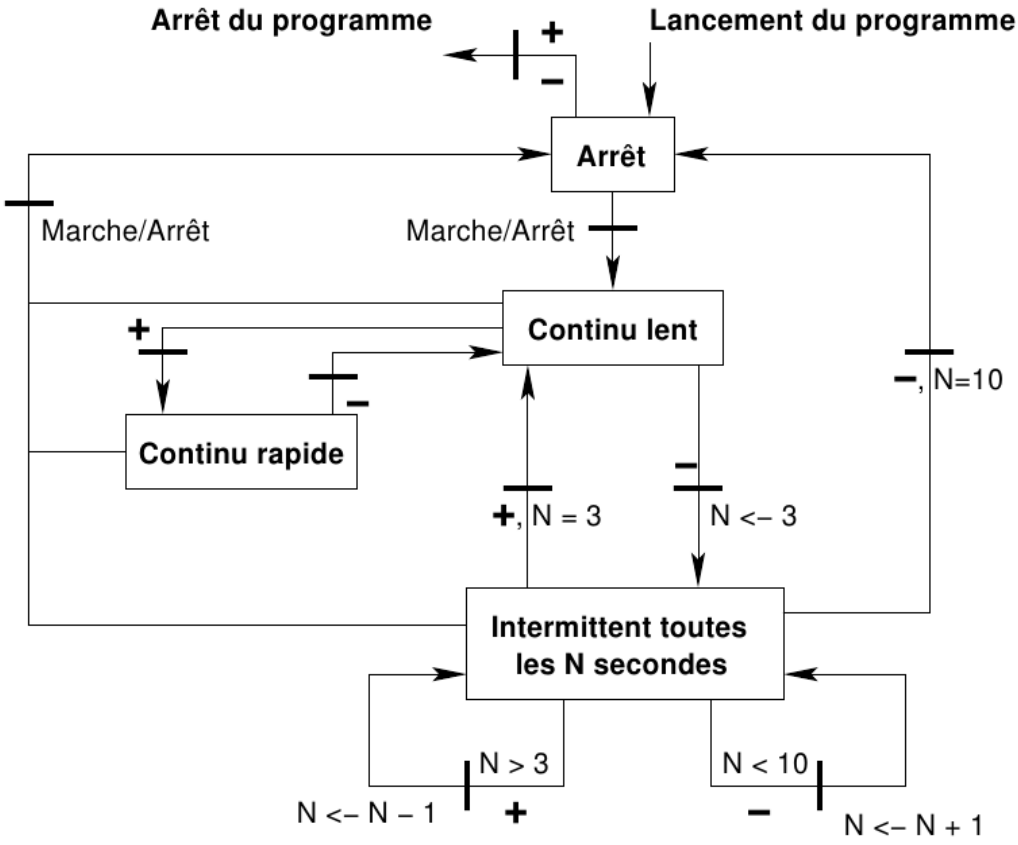
Communication inter-processus

On veut gérer un moteur d'essuie-glace capable de fonctionner soit en continu (2 vitesses possibles), soit en intermittent. Le moteur est piloté par quatre commandes : continu lent, continu rapide, impulsion et Marche/Arrêt.



Il s'agit donc d'écrire le programme pilotant ce moteur (PILOTE) qui admet 4 commandes : Marche (M), Arrêt (A), + (pour augmenter la vitesse), - (pour diminuer la vitesse).

Le fonctionnement du pilote et le rôle des commandes est décrit par le diagramme état/transition suivant :



Au lancement du programme, le moteur est à l'arrêt. L'appui sur la touche Marche le fait passer en mode "Continu lent". Les passages du mode "Continu lent" au mode "Continu rapide" et réciproquement se font respectivement par l'appui sur la touche + et par l'appui sur la touche -.

Du mode "Continu lent", il est possible de passer au mode "Intermittent" par l'appui sur la touche -. Dans ce mode, le moteur se met en marche toutes les 3 secondes. Il est possible d'ajuster la vitesse du mode "Intermittent" à l'aide des touches + pour augmenter la vitesse d'une seconde, et – pour ralentir la vitesse d'une seconde.

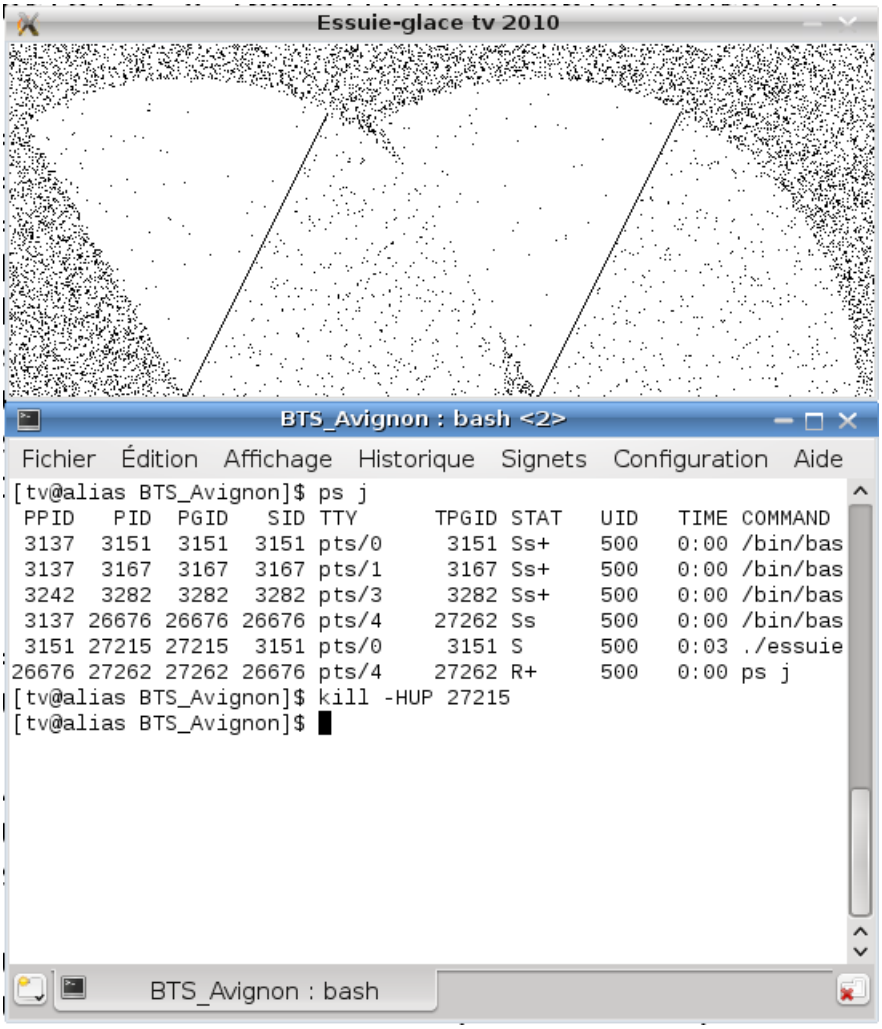
En deçà de 3s, le moteur se remet en mode continu lent et au delà de 10 secondes, il s'arrête.

L'appui sur la touche Quitter (Q) dans l'état d'arrêt met fin au programme.

On va simuler ce fonctionnement en utilisant les interruptions logicielles d'Unix (signaux). Nous allons utiliser 4 signaux : 3 générés par le clavier pour simuler l'appui sur une commande du pilote et l'horloge pour la programmation du mode intermittent. Les touches M, A, +, - et Q sont donc ici remplacées par l'envoi de signaux :

Identificateur	Libellé	Simulation de	Généré par
SIGINT	interrupt	Marche/Arrêt	Ctrl-C
SIGQUIT	quit	+	Ctrl-X
SIGTSTP	keyboard stop	-	Ctrl-Z
SIGALRM	alarm clock	intermittent	alarm(N)

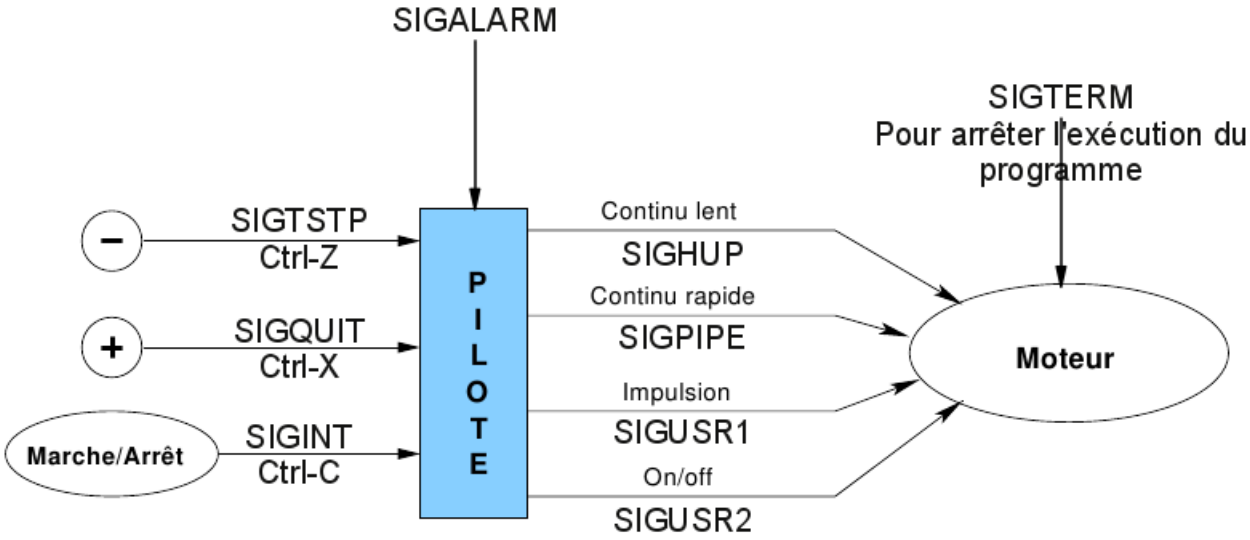
On vous fournit également un programme de simulation du moteur, qui affiche une fenêtre sur l'écran avec deux essuie-glace et une simulation de la pluie (programme réalisé par Sylvain Viart en 96/97 pour XWindow et que j'ai adapté pour la SDL).



On utilise donc deux simulateurs pour valider le programme PILOTE :

- le clavier qui génère les signaux pour simuler les 4 commandes : Marche (M), Arrêt (A), + (pour augmenter la vitesse), - (pour diminuer la vitesse)
- le logiciel graphique essuie-glace qui reçoit les signaux en provenance du programme de commande

On vous demande de compléter le programme pilote.c fourni afin qu'il assure la gestion des signaux suivantes :



9) Compléter les parties /* TODO */ dans le source pilote.c

10) Configurer le clavier pour la gestion des signaux destinés au pilote :

```
$ stty intr "^C" quit "^X" susp "^Z"
```

11) Tester.