# Programmation système

## Ressource R3.05 - Signaux

monnerat@u-pec.fr

IUT de Fontainebleau

# Sommaire

1. Rappels de C

2. Généralités sur les signaux

3. Implantation

4. API - Primitives de base

5. Race conditions

6. `SIGCHLD`

# Rappels de C

# Rappels de C

## Pointeur de fonctions, type `void *`

En C, l'identitiant d'une fonction est son adresse (son point d'entrée) en mémoire.

- Il est possible de manipuler les adresses des fonctions en C, et d'avoir des pointeurs sur des fonctions.
- Il est obligatoire de savoir manipuler les pointeurs sur les fontions pour gérer les signaux et les threads.

Le type d'un pointeur sur fonction doit contenir les types des paramètres de la fonction, et le type de retour.

- les paramètres n'ont pas besoin d'avoir de nom,
- le compilateur doit juste savoir quel type empiler sur la pile

Pour déclarer un pointeur sur une fonction :

```
type_retour (*nompointeur) (liste_arguments...);
```

Exemple :

```
int (*fct)(int);
```

déclare `fct` comme un pointeur sur une fonction prenant en argument un entier, et renvoyant un entier. Appeler une fonction par un pointeur se fait de la même manière que pour une fonction normale.

```c
#include <stdio.h>
int carre (int x) {
  return x*x;
}
int cube (int x) {
  return x*x*x;
}
void boucle ( int (*f) (int)) {
  int i ;
  for (i=1; i <=10; i++)
    printf ("%d : %d\n",i, f(i)); // equivalent à (*f)(i)
}
int main(){
  boucle(&carre);  // equivalent à boucle(carre)
  boucle(&cube);
}
```

# Passage de paramètre générique à une fonction

Beaucoup de fonctions de librairies standards du C utilise le type `void *`.

```
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
void bzero(void *s, size_t n);
void qsort(void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *));
```

- Il s'agit d'un pointeur (adresse) "générique" (sans type), qui permet de transmettre à une fonction une adresse de n'importe quelle type : `char *`,`int *`, `int **`, `etc.`
- La manière d'interpréter cette adresse est laissée "libre".
- Normalement, il est interdit de faire de l'arithmétique sur un pointeur `void *`. `gcc` le permet (`-Wpointer-arith` donnera une erreur).

# Exemple 1 : memcpy()

```
void *
memcpy (void *dest, const void *src, size_t len)
{
  char *d = dest;
  const char *s = src;
  while (len--)
    *d++ = *s++;
  return dest;
}
```

# Exemple 2 : `atexit()`

```c
#include <stdlib.h>
int atexit(void (*function)(void));
```

`atexit()` enregistre une fonction qui sera appelée à la fin (normale) du processus.

# Exemple 3 : `qort()`

```c
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *));
```

Le troisième argument représente la fonction de comparaison (ordre) que `qsort()` doit utiliser. pour trier le tableau dans l'ordre croissant.

# Généralités sur les signaux

# Généralités sur les signaux

## Définition

# Les signaux

C'est un mécanisme de communication asynchrone.

"Interruption logicielle". Fonction ?

- Avertir un processus d'un evénement.
- Exécuter un gestionnaire.

Ils sont représentés par un numéro et un nom symbolique dans le header `signal.h`

Extrait du header

```
#define SIGBUS      7
#define SIGFPE      8
#define SIGKILL     9
#define SIGUSR1    10
```

Qui peut les générer ?

- un utilisateur : depuis un terminal de contrôle (ctrl+C,ctrl+Z), commande shell `kill`.
- un processus.
- le noyau (instruction invalide, violation mémoire, etc.)

# Emission et délivrance

Deux phases distinctes !

1. Émission du signal
   Le noyau sauvegarde (vecteur de bits dans le PCB du processus) le signal émis jusqu'à ce qu'il puisse être délivré.
2. Délivrance du signal.
   Le noyau **force** le processus destinataire à trairer le signal. Les processus peuvent :
   - ignorer le signal.
   - bloquer le signal.
   - capter et traiter le signal avec la fonction gestionnaire par défaut, ou celle qu'ils ont enregistrée préalablement.

`SIGKILL` et `SIGSTOP` ne peuvent pas être captés, bloqués ou ignorés !

| Signal | Action | Commentaire |
|---|---|---|
| SIGHUP | A | Déconnexion sur terminal de contrôle, ou mort du processus leader |
| SIGINT | A | Interruption depuis le clavier. |
| SIGQUIT | A | Demande 'Quitter' depuis le clavier. |
| SIGILL | A | Instruction illégale. |
| SIGABRT | C | Signal d'arrêt depuis abort(3). |
| SIGFPE | C | Erreur mathématique virgule flottante. |
| SIGKILL | AEF | Signal 'KILL'. |
| SIGSEGV | C | Référence mémoire invalide. |
| SIGPIPE | A | Ecriture dans un tube sans lecteur. |
| SIGALRM | A | Temporisation alarm(2) écoulée. |
| SIGTERM | A | Signal de fin. |
| SIGUSR1 | A | Signal utilisateur 1. |
| SIGUSR2 | A | Signal utilisateur 2. |
| SIGCHLD | B | Fils arrêté ou terminé. |
| SIGCONT | | Continuer si arrêté. |
| SIGSTOP | DEF | Arrêt du processus. |
| SIGTSTP | D | Stop invoqué depuis tty. |
| SIGTTIN | D | Lecture sur tty en arrière-plan. |
| SIGTTOU | D | Ecriture sur tty en arrière-plan. |

| | |
|---|---|
| A | Par défaut, terminer le processus |
| B | Par défaut, ignorer le signal |
| C | Par défaut, créer un fichier core |
| D | Par défaut arrêter le processus |
| E | Le signal ne peut pas être intercepté |
| F | Le signal ne peut pas être ignoré |

# Implantation

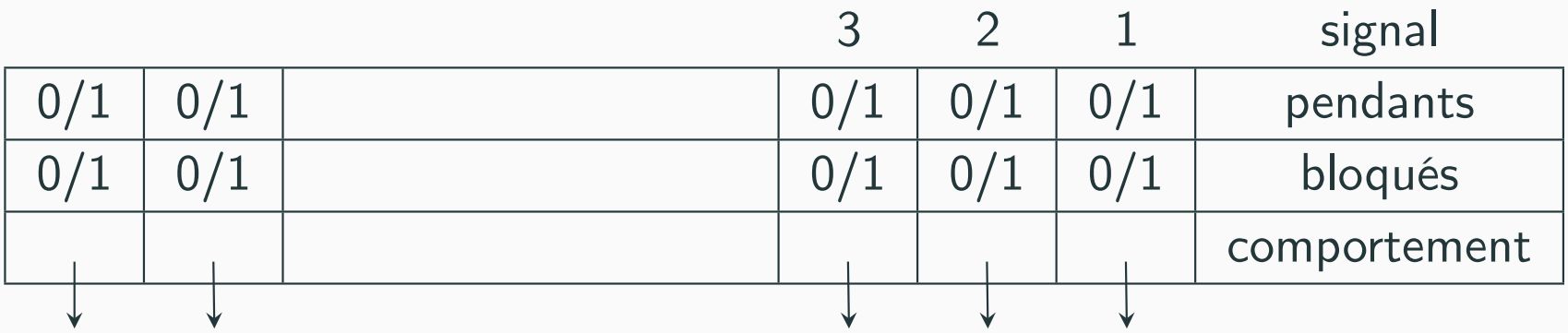
# Implantation

## Informations

# Informations associées aux signaux

Le noyau maintient pour chaque processus (dans son bloc de contrôle) les informations nécessaires à la gestion des signaux.

- un vecteur de bits indiquant les signaux bloqués.
- un vecteur de bits indiquant les signaux en attente (pending signal). (pas de mémorisation du nombre d'occurences !)
- un vecteur de gestionnaires indiquant les actions à déclencher lors de la prise en compte des signaux :
  - le traitement par défaut
  - ignorer
  - un gestionnaire propre



# Héritage

Que se passe-t-il avec fork (fils) et exec (écrasement) ?

Avec fork :

- hérite les signaux bloqués
- hérite les traitements de signaux
- n'hérite pas les signaux pendants (vide).

Avec `exec`

- hérite les signaux bloqués
- hérite les signaux pendants
- n'hérite pas les dispositions de traitement (défaut).

| | `fork()` | `exec()` |
|---|---|---|
| Disposition | hérité | non hérité |
| bloqué | hérité | hérité |
| pendant | non hérité | hérité |

# Implantation

## Prise en compte d'un signal

Lors de l'émission du signal, le noyau vérifie les permissions.

Si le signal est ignoré, le noyau ne fait rien.

Sinon, le signal est marqué en attente (pendant) s'il ne l'était pas. Un processus dans l'état `TASK_INTERRUPTIBLE` est réveillé.

Lorsqu'un processus redevient actif en mode utilisateur (retour appel système, interruption), le noyau vérifie l'existence de signaux pendants, et les traite tous (sauf ceux qui sont bloqués)

Remarques :

- Un processus endormi sur un appel système est reveillé. Certaines primitives sont automatiquement relancées une fois le signal traité.
- Les traitements de `SIGKILL` et `SIGCONT` sont particuliers.

# API - Primitives de base

# API - Primitives de base

## Envoi

# Envoi : primitive kill()

```c
#include <signal.h>

int kill(pid_t pid, int sig);
```

Envoie le signal `sig` à

- `pid > 0` : le processus pid.
- `pid == 0` : le groupe de l'appelant.
- `pid == -1` : tous les processus (si les droits le permettent).
- `pid < -1` : le groupe dont le pid vaut la valeur absolu de l'argument.

# Envoi : primitive `alarm()`

```c
#include <unistd.h>

unsigned alarm(unsigned seconds);
```

Programme une alarme pour le processus appelant. `SIGALARM` lui sera envoyé au bout de `seconds`.

- Les alarmes ne sont pas empilées ! `alarm()` remplace toute alarme précédente.
- En particulier, si seconds == 0, si `SIGALARM` est pendant, il est nettoyé.

Attention : pour un signal périodique, il faudra réarmer manuellement.

# Envoi : primitive `setitimer()`

```c
#include <sys/time.h>
int getitimer(int which, struct itimerval *curr_value);
int setitimer(int which,
  const struct itimerval *restrict new_value,
  struct itimerval *_Nullable restrict old_value);

struct itimerval {
    struct timeval it_interval;  // période
    struct timeval it_value;     // premier déclenchement
};

struct timeval {
    time_t      tv_sec;          // secondes
    suseconds_t tv_usec;         // microsecondes
};
```

Si `it_interval` est nul, le timer est one-shot, comme `alarm`.

Le premier argument de `setitimer()` indique ce que mesure le timer.

| Timer | Mesure | Signal |
|---|---|---|
| `ITIMER_REAL` | temps réel écoulé | `SIGALRM` |
| `ITIMER_VIRTUAL` | temps CPU du processus en mode utilisateur | `SIGVTALRM` |
| `ITIMER_PROF` | temps CPU utilisateur + noyau | `SIGPROF` |

# API - Primitives de base

## Masquage

# Blocage `sigprocmask()`

Manipulation d'un ensemble de signaux :

```c
#include <signal.h>

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);
```

Bloquer ou débloquer un ensemble de signaux

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

how :

- SIG_SETMASK : nouveau mask = set
- SIG_BLOCK : nouveau mask = ancien $\cup$ set,
- SIG_UNBLOCK : nouveau mask = ancien$\backslash$set

A la reception d'un signal bloqué :

- le signal est marqué pendant (mis en attente).
- il sera traité s'il est débloqué.

Par défaut, tout signal est bloqué pendant l'éxécution de son gestionnaire.

# API - Primitives de base

## Traitement

# Lister les signaux pendants

```c
#include <signal.h>

int sigpending(sigset_t *set);
```

- Copie dans set la liste des signaux pendants.
- C'est particulièrement utile si des signaux sont masqués (et non ignorés).

# Traitement : sigaction()

```
int sigaction(int sig,
  const struct sigaction *restrict act,
  struct sigaction *restrict oact);
```

permet de modifier le traitement associé à un signal :

- `sig` : le signal.
- `act`/`oact` : nouveau comportement/sauvegarde de l'ancien.

La structure sigaction :

- `sa_handler` : le gestionnaire (pointeur de fonction)
- `sa_mask` : ensemble additionnel de signaux bloqués.
- `sa_flags` : options.

L'option `SA_RESTART` relancera automatiquement les appels systèmes interruptibles interrompus par le signal.

# API - Primitives de base

## Attente de signaux

# Attente de signaux

`pause()`

```c
#include <unistd.h>
int pause(void);
```

`sigsuspend()`

```c
#include <signal.h>
int sigsuspend(const sigset_t *sigmask);
```

- `pause()` met le processus en pause, jusqu'à ce qu'un signal qui exécute un handler arrive (le handler est exécuté en premier).
- Problème : un signal non masqué peut arriver avant l'appel à pause(), et être "perdu" ...
- `sigsuspend()` change temporairement le masque des signaux masqués, et attend jusqu'à ce qu'un signal arrive.

# Signaux et appels systèmes

- Certains appels systèmes peuvent être interrompus par un signal. Dans ce cas, l'appel système échoue, et le code retour (`errno`) sera `EINTR`. Utiliser l'option

  `SA_RESTART`

  permet de relancer l'appel système après le traitement du signal.
- Lors d'un fork, les signaux pendants ne sont pas hérités (le masque et les handlers, si)
- Lors d'un exec, les handlers ne sont pas hérités (le masque et les signaux pendants, si)

# Fonctions `async-signal-safe`

Un handler peut interrompre le code principal à n'importe quel instant, y compris en plein milieu d'une fonction de la libc en train de modifier ses propres structures internes. Si le handler rappelle cette même fonction, il peut la retrouve dans un état incohérent ((non-)réentrance).

POSIX définit alors un ensemble restreint de fonctions pouvant être utilisées de manière sûre dans ce contexte.

Exemples de fonctions

- sûres : `write`, `read`, `kill`, `exit`, `signal`, `sigaction`, `alarm`, `waitpid`.
- pas sûres : `printf`, `malloc`, `free`, `rand`, etc.

Pour une liste exhaustive

```
man 7 signal-safety
```

# Race conditions

Problème : mise en attente d'un fils d'un signal (SIGUSR1) du père.

Première idée : Idée Naïve

```c
void signal_handler(int n){}
int main(){
  pid_t p;
  set_signal_handler(SIGUSR1,signal_handler);
  p=fork();
  if (p>0) {
    kill(p,SIGUSR1);
    printf("BYE\n");
    exit(1);
  }
  pause();
  printf("DONE\n");
}
```

Ca fonctionne ? pourquoi ?

## Deuxième idée

```c
static volatile sig_atomic_t got_signal=0;
void signal_handler(int n){
  got_signal = 1;
}
int main(){
  pid_t p;
  p=fork();
  if (p>0) {
    /* .... */
  }
  if (!got_signal)
    pause();
  printf("DONE\n");
}
```

Ca fonctionne ? pourquoi ?

## Utilisation de sigsuspend

```
void signal_handler(int n){}
int main(){
  pid_t p;
  sigset_t set,oset;
  set_signal_handler(SIGUSR1,signal_handler);
  sigaddset(&set,SIGUSR1);
  sigprocmask(SIG_SETMASK,&set,&oset);
  p=fork();
  if (p>0) {    /*  ... */ }
  sigsuspend(&oset);
  printf("DONE\n");
}
```

- SIGUSR1 est bloqué dans le fils.
- sigsuspend, atomiquement, débloque SIGUSR1, et attend un signal. Si le signal a déjà été envoyé, il sera pris en compte, sinon, attente.

# SIGCHLD

# Traitement de SIGCHLD

Lorsqu'un processus fils se termine avant que son père n'ait fait appel à `wait`, le noyau maintient des informations (comme son code de retour) pour que son père puisse le faire ultérieureument. On parle d'état zombie (le fils ne s'exécute plus, mais consomme quand même des ressources).

On peut traiter explicitement le signal :

```
void handlerSigchld(int n){
  while(waitpid(-1,NULL,WNOHANG)>0);
}
int main(){
  struct sigaction act={0};
  act.sa_handler = handlerSigchld;
  sigemptyset(&act.sa_mask);
  sigaction(SIGCHLD,&act,NULL);
  /* .... */
}
```

Expliquez la boucle dans le handler.

La norme POSIX.1-2001 (Linux y adhére) permet au parent de choisir que le noyau récupère automatiquement les processus enfants qui se terminent en positionnant explicitement le traitement à `SIG_IGN`.

```
struct sigaction sa = {0};
sa.sa_handler = SIG_IGN;
sigemptyset(&sa.sa_mask);
sigaction(SIGCHLD,&sa,NULL);
```

On peut également utiliser le flag `NO_NOCLDWAIT` :

```
struct sigaction sa = {0};
sa.sa_flags = SA_NOCLDWAIT;
sigaction(SIGCHLD,&sa,NULL) ;
```