

---

# Gestion des signaux

---

Louis Mandel

`louis.mandel@lri.fr`

Laboratoire de Recherche en Informatique

Université Paris-Sud 11

IFIPS

Cycle ingénieur de la filière étudiant

année 2007/2008

# Les signaux

---

- ▶ Signaux
  - ▶ Événements externes qui changent le déroulement d'un programme, de manière asynchrone
  - ▶ Émis par un autre processus ou par le système
- ▶ Comportement à la réception d'un signal (selon le signal et les réglages) :
  - ▶ Terminaison de l'exécution
  - ▶ Suspension de l'exécution (le processus père est prévenu)
  - ▶ Rien : le signal est ignoré
  - ▶ Exécution d'une fonction définie par l'utilisateur

## Les principaux signaux

---

Nom	Signification	Comportement
SIGHUP	Hang-up (fin de connexion)	T(erminaison)
SIGINT	Interruption (Ctrl-c)	T
SIGQUIT	Interruption forte (Ctrl-\)	T + core
SIGFPE	Erreur arithmétique	T + core
SIGKILL	Interruption immédiate et absolue	T + core
SIGSEGV	Violation des protections mémoire	T + core
SIGPIPE	Écriture sur un pipe sans lecteurs	T
SIGTSTP	Arrêt temporaire(Ctrl-z)	Suspension
SIGCONT	Redémarrage d'un fils arrêté	Ignoré
SIGCHLD	un des fils est mort ou arrêté	Ignoré

## Les principaux signaux

---

Nom	Signification	Comportement
SIGALRM	Interruption d'horloge	Ignoré
SIGTSTP	Arrêt temporaire(Ctrl-z)	Suspension
SIGUSR1	Émis par un processus utilisateur	T
SIGUSR2	Émis par un processus utilisateur	T

► Liste complète des signaux : `man 7 signal`

- ▶ Message associé à un signal
  - ▶ (non standard, ni POSIX, ni ANSI C)

```
#include <signal.h>
#include <string.h>
char *strsignal(int sig)
extern const char *const sys_signame[NSIG];
void psignal(int sig, const char *s);
```

- ▶ Exemple :

```
int main () {
    int signum;
    for (signum=0; signum<NSIG; signum++) {
        fprintf(stderr, "(%2d) %8s : %s\n",
            signum, sys_signame[signum], strsignal(signum));
    }
    return 0; }
```

# Attente d'un signal

---

- ▶ Primitive bloquante
  - ▶ `#include <unistd.h>`  
`int pause(void);`
  - ▶ bloquante jusqu'à la délivrance d'un signal
  - ▶ puis action en fonction du comportement associé au signal
- ▶ Attention : sans pause la délivrance d'un signal déclenche aussi le comportement associé au signal.

# Envoie de signaux

---

- ▶ La commande `kill`
  - ▶ `kill [-signal_name] pid ...`  
`kill [-signal_number] pid ...`
    - ▷ numéros normalisés
    - ▷ nécessite des droits

## Exemple :

(await.c)

```
int main() {  
    fprintf(stderr, "[%d] pausing...\n", getpid());  
    pause();  
    fprintf(stderr, "[%d] terminating...\n", getpid());  
    return 0;  
}
```

### ► Exemple :

```
--> kill -0 16277  
--> kill -KILL 16277  
--> kill -0 16277  
bash: kill: (16277) - No such process  
--> kill -9 16304  
--> kill -USR1 16312
```



## Envoie de signaux

---

- ▶ Un processus envoie un signal à un autre processus désigné
  - ▶ `#include <signal.h>`
  - `int kill(pid_t pid, int sig);`
  - ▶ retourne -1 en cas d'erreur
  - ▶ signal de numéro 0 : pas de signal : test de validité de pid
- ▶ Un processus envoie un signal à lui même
  - ▶ `int raise(int sig);`
    - ▷ équivalent à : `kill(getpid(), sig);`
  - ▶ `int abort(void);`
    - ▷ équivalent à : `raise(SIGABRT);`

## Exemple :

(kill.c)

```
int main() {
    pid_t pid; int statut;
    printf("Lancement du processus %d\n", getpid());
    switch (pid = fork()) {
    case -1: exit(1);
    case 0: while(1) sleep(1); exit(1);
    default:
        printf("Processus fils %d cree\n", pid); sleep(10);
        if ( kill(pid,0) == -1 ) printf("fils %d inaccessible\n", pid);
        else {
            printf("Envoi du signal SIGUSR1 au processus %d\n", pid);
            kill(pid, SIGUSR1);
        }
        pid = waitpid(pid, &statut, 0);
        printf("Statut final du fils %d : %d\n", pid, statut); } }
```

# États d'un signal

---

- ▶ Un signal est **envoyé**
  - ▶ par un processus émetteur à un processus destinataire
- ▶ Un signal est **pendant** (*pending*)
  - ▶ tant qu'il n'a pas été traité par le processus destinataire
- ▶ Un signal est **délivré**
  - ▶ lorsqu'il est pris en compte par ce processus destinataire
- ▶ Pourquoi un état pendant ?
  - ▶ le signal peut être bloqué (masqué, retardé) par le processus destinataire
  - ▶ sera délivré quand il sera débloqué
  - ▶ un signal est bloqué durant l'exécution du traitement d'un signal de même type
  - ▶ il ne peut exister qu'un signal pendant d'un type donné
  - ▶ **des signaux peuvent être perdus**

# Réglage du comportement à la réception d'un signal

---

- ▶ Différents réglages possibles pour chaque type de signal
  - ▶ comportement par défaut
  - ▶ ignorance
  - ▶ traitement personnalisé
  - ▶ masquage (blocage)
- ▶ Comportement par défaut
  - ▶ identifié par la valeur symbolique SIG\_DFL
  - ▶ traitement propre à chaque type de signal : terminaison, ignorance, suspension, etc.
- ▶ Ignorance d'un signal
  - ▶ identifié par la valeur symbolique SIG\_IGN
  - ▶ le signal est délivré, mais le comportement est de ne rien faire

# Réglage du comportement à la réception d'un signal

---

- ▶ Traitement personnalisé
  - ▶ exécuté par le processus destinataire du signal
  - ▶ Le prototype de la fonction de traitement est :  
`void handler(int signom)`  
donc de type `void (*phandler) (int)`
    - ▷ paramètre : numéro du type de signal
  - ▶ Retour au code interrompu après l'exécution de la fonction de traitement du signal
    - ▷ Attention : le retour après le traitement d'un signal SIGSEGV, SIGILL ou SIGFPE est dangereux (le problème doit être résolu)
- ▶ Masquage de signaux
  - ▶ pour chacun des signaux indiquer si on le bloque ou non

# Manipulation d'ensembles de signaux

---

- ▶ Type ensemble de signaux
  - ▶ `sigset_t`
  - ▶ défini dans `<signal.h>`
- ▶ Initialisation
  - ▶ à vide

```
int sigemptyset(sigset_t *psigset);
```
  - ▶ à plein

```
int sigfillset(sigset_t *psigset);
```
- ▶ Ajout et suppression
  - ▶ 

```
int sigaddset(sigset_t *psigset, int sig);
```

```
int sigdelset(sigset_t *psigset, int sig);
```
- ▶ Test d'appartenance
  - ▶ 

```
int sigismember(sigset_t *psigset, int sig);
```

# Installation d'un masque de blocage

---

- ▶ Installation manuelle d'un nouveau masque

- ▶ `int sigprocmask(int op, sigset_t *new, sigset_t *old);`

- ▶ Le paramètre `op` détermine le nouvel ensemble :

op	nouveau masque
SIG_SETMASK	*new
SIG_BLOCK	*new    *old
SIG_UNBLOCK	*old - *new

- ▶ récupère l'ancien masque dans `old`

- ▶ Liste des signaux pendants masqués

- `int sigpending(sigset_t *pending);`

## Exemple :

masque.c

---

```
sigset_t ens1, ens2; int sig;
int main () {
    sigemptyset(&ens1);
    sigaddset(&ens1, SIGINT); sigaddset(&ens1, SIGQUIT); sigaddset(&ens1, SIGUSR1);
    sigprocmask(SIG_SETMASK, &ens1, NULL);
    raise(SIGINT); kill(getpid(), SIGINT); kill(getpid(), SIGUSR1);
    sigpending(&ens2);
    printf("Signaux pendants : ");
    for(sig = 1; sig < NSIG; sig++)
        if (sigismember(&ens2, sig)) printf("%d ", sig);
    putchar('\n'); sleep(10);
    sigemptyset(&ens1);
    printf("Deblocage de tous les signaux\n");
    sigprocmask(SIG_SETMASK, &ens1, NULL);
    printf("Fin du processus\n"); exit(0); }
```



## Traitant de signal personnalisé

---

- ▶ La primitive `sigaction`
  - ▶ `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
  - ▶ installe le traitant `new`
  - ▶ récupère l'ancien traitant dans `old`

# Traitant de signal personnalisé

---

- ▶ La structure sigaction
  - ▶ struct sigaction {  
    void (\*sa\_handler) (int);  
    sigset\_t sa\_mask;  
    int sa\_flags;  
}
  - ▶ la fonction sa\_handler est le traitement
    - ▷ cette fonction peut être égale à SIG\_DFL ou SIG\_IGN
  - ▶ les signaux sa\_mask seront masqués durant l'exécution de la fonction
  - ▶ sa\_flags : options (cf. man sigaction)

```
#define NSIGMAX 5
void set_default() {
    struct sigaction sa;
    sa.sa_handler = SIG_DFL;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGINT, &sa, NULL);
}
void int_handler(int sig) {
    static int nsig = 0;
    if (nsig++ < NSIGMAX) printf(" C-c won't kill me\n");
    else { printf(" unless you insist...\n"); set_default(); }
}
int main () {
    struct sigaction sa;
    sa.sa_handler = int_handler; sigemptyset(&sa.sa_mask); sa.sa_flags = 0;
    sigaction(SIGINT, &sa, NULL);
    for(;;) pause();
    fprintf(stderr, "bye\n");
    return 0; }
```

## ► Possible perte de signaux

```
void handler(int sig) {
    static int nusr1 = 0;
    switch (sig) {
        case(SIGUSR1): nusr1++; break;
        case SIGINT: printf("signaux recus: %d\n", nusr1); exit(EXIT_SUCCESS);
        default: ; }
}

int main (int argc, char *argv[]) {
    struct sigaction sa; int nsig, i;
    nsig = atoi(argv[1]);
    sa.sa_handler = handler; sigemptyset(&sa.sa_mask); sa.sa_flags = 0;
    sigaction(SIGINT, &sa, NULL);  sigaction(SIGUSR1, &sa, NULL);
    switch(fork()) {
        case 0:
            for(i=0; i<nsig; i++) kill(getppid(), SIGUSR1);
            printf("bye\n"); exit(EXIT_SUCCESS);
        default: for(;;) sleep(1);  }
    exit(EXIT_SUCCESS);
}
```

► Correction de faute nécessaire

```
void handler(int sig) {
    fprintf(stderr, "I will not core dumped...\n");
}

int main () {
    char *c;
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGSEGV, &sa, NULL);
    for (c = ((char *) sbrk(0)); ; c++) *c = 'a';
    fprintf(stderr, "bye\n");
    exit(EXIT_SUCCESS);
}
```

# Temporisation

---

- ▶ Interrompre le processus au bout d'un délai
  - ▶ réception d'un signal SIGALRM à l'expiration du délai
  - ▶ requête au système de délivrance du signal
- ▶ Armement du minuteur
  - ▶ `#include <unistd.h>`  
`unsigned int alarm(unsigned int seconds);`
  - ▶ un seul minuteur par processus
  - ▶ un nouvel armement annule le précédent
  - ▶ délai nul supprime la requête

```
#define LINE_MAX 128
#define DELAY 10
void beep(int sig) { printf("\ntrop tard..\n"); }
int main () {
    struct sigaction sa;
    char answer[LINE_MAX];
    sa.sa_handler = beep; sigemptyset(&sa.sa_mask); sa.sa_flags = 0;
    sigaction(SIGALRM, &sa, NULL);
    printf("Reponse ? ");
    alarm(DELAY);
    if (fgets(answer, LINE_MAX, stdin)) {
        alarm(0);
        printf("ok...\n");
    }
    exit(EXIT_SUCCESS); }
```

# Temporisations avancées

---

- ▶ Temporisation par alarm
  - ▶ temps-réel (*wall-clock time*)
  - ▶ résolution à la seconde
- ▶ Temporisation par setitimer (<sys/time.h>)
  - ▶ `int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);`

- ▷ Trois temporisations

which	temporisation	signalisation
ITER_REAL	temps réel	SIGALRM
ITER_VIRTUAL	temps processeur en mode utilisateur	SIGVTALRM
ITER_PROF	temps processeur total	SIGPROF

- ▷ retourne l'ancien minuteur
- ▶ Résolution de la granularité des durées au mieux de l'implantation



# Temporisations avancées

---

- ▶ La structure `timeval`

- ▶ 

```
struct timeval {  
    time_t    tv_sec;           /* seconds */  
    long int  tv_usec;         /* microseconds */  
};
```

- ▶ La structure `itimerval`

- ▶ 

```
struct itimerval {  
    struct timeval it_interval; /* timer interval */  
    struct timeval it_value;    /* current value */  
};
```

- ▶ timer périodique
    - ▶ spécifie une échéance à `it_value`
    - ▶ puis toutes les `it_interval`
    - ▶ `it_value` à 0 : annulation
    - ▶ `it_interval` à 0 : pas de réarmement

```
static struct tms start, end;
static float tics_to_seconds(clock_t tics) {
    return tics/(float)sysconf(_SC_CLK_TCK); }
void handler(int sig) {
    times(&end);
    printf("%.6f\n", tics_to_seconds(end.tms_utime - start.tms_utime));
    times(&start); }
int main () {
    struct itimerval itv; int i; struct sigaction sa;
    sa.sa_handler = handler; sigemptyset(&sa.sa_mask); sa.sa_flags = 0;
    sigaction(SIGVTALRM, &sa, NULL);
    itv.it_value.tv_sec = 0 ; itv.it_value.tv_usec = 200000;
    itv.it_interval.tv_sec = 0 ; itv.it_interval.tv_usec = 500000;
    times(&start); setitimer(ITIMER_VIRTUAL, &itv, NULL);
    for (;;) i++; exit(EXIT_SUCCESS); }
```

## Terminaison / blocage des fils

---

- ▶ Le processus père est prévenu par signal de la terminaison d'un de ses fils
  - ▶ signal SIGCHLD
  - ▶ comportement par défaut : ignorance
  - ▶ traitant de signal typique :
    - ▷ élimination du processus zombi
    - ▷ appel `wait` / `waitpid`
- ▶ Processus père est prévenu par signal de l'arrêt d'un des ses fils
  - ▶ signal SIGCHLD
  - ▶ comportement par défaut : ignorance
  - ▶ passage du fils dans l'état bloqué par réception de SIGSTOP ou SIGTSTP
  - ▶ le relancer par un signal SIGCONT

```
static pid_t fils;
static void handler(int sig) {
    printf("[%d] ai reçu signal %d\n", getpid(), sig);
    kill(fils, SIGCONT);
}
int main() {
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGCHLD, &sa, NULL);
    if ((fils = fork()) == 0) { /* fils */
        printf("[%d] kill(%d, %d)\n", getpid(), getpid(), SIGSTOP);
        kill(getpid(), SIGSTOP);
        kill(getpid(), SIGSTOP);
        exit(EXIT_SUCCESS);
    }
    /* pere */
    for(;;) pause();
    exit(EXIT_SUCCESS);
}
```

```
static pid_t fils;
static void handler(int sig) {
    printf("[%d] ai reçu signal %d\n", getpid(), sig);
    system("pwd");
}
int main() {
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGCHLD, &sa, NULL);
    if ((fils = fork()) == 0) { /* fils */
        printf("[%d] kill(%d, %d)\n", getpid(), getpid(), SIGSTOP);
        kill(getpid(), SIGSTOP);
        kill(getpid(), SIGSTOP);
        exit(EXIT_SUCCESS);
    }
    /* pere */
    for(;;) pause();
    exit(EXIT_SUCCESS);
}
```

```
static struct sigaction sa;

static void handler(int sig) {
    printf("[%d] ai reçu signal %d\n", getpid(), sig);
    sa.sa_handler = SIG_DFL;
    sigaction(SIGCHLD, &sa, NULL);
    system("pwd");
    sa.sa_handler = handler;
    sigaction(SIGCHLD, &sa, NULL);
    kill(fils, SIGCONT);
}
```

## Contrôle du point de reprise

---

- ▶ Reprise au retour d'un traitant de signal
  - ▶ le code du processus là où il a été interrompu
  - ▶ en général...
- ▶ Signal dans un appel système interruptible
  - ▶ exemple : `read`, `wait`, `system`, etc.
  - ▶ l'appel système est interrompu
  - ▶ et non repris
  - ▶ il retourne, typiquement, `-1`
  - ▶ `errno` est positionnée à `EINTR`
  - ▶ c'est au programme de le relancer

```
static void handler(int sig) {
    pid_t pidz;
    int status;
    pidz = wait(&status);
    if (pidz == -1) { perror("handler wait"); exit(EXIT_FAILURE); }
    printf("handler wait: pidz %d, status %d\n", pidz, WEXITSTATUS(status)); }

int main() {
    struct sigaction sa;
    pid_t pidz;
    int status;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGCHLD, &sa, NULL);
    if (fork() == 0) { /* fils */
        sleep(5); exit(2);
    }
    pidz = wait(&status);
    if (pidz == -1) { perror("main wait"); exit(EXIT_FAILURE); }
    printf("main wait: pidz %d, status %d\n", pidz, WEXITSTATUS(status));
    exit(EXIT_SUCCESS); }
```



## Contrôle du point de reprise

---

- ▶ Reprise possible des appels systèmes interrompus
  - ▶ drapeau SA\_RESTART dans struct sigaction
  - ▶ 

```
struct sigaction sa;  
sa.sa_handler = ... ;  
sa.sa_mask = ... ;  
sa.sa_flags = SA_RESTART;  
sigaction(..., &sa, ...);
```

```
static void handler(int sig) {
    pid_t pidz;
    int status;
    pidz = wait(&status);
    if (pidz == -1) { perror("handler wait"); exit(EXIT_FAILURE); }
    printf("handler wait: pidz %d, status %d\n", pidz, WEXITSTATUS(status)); }

int main() {
    struct sigaction sa;
    pid_t pidz;
    int status;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;          /* XXX */
    sigaction(SIGCHLD, &sa, NULL);
    if (fork() == 0) { /* fils */
        sleep(5); exit(2);
    }
    pidz = wait(&status);
    if (pidz == -1) { perror("main wait"); exit(EXIT_FAILURE); }
    printf("main wait: pidz %d, status %d\n", pidz, WEXITSTATUS(status));
    exit(EXIT_SUCCESS); }
```