La Gestion des Signaux

Dans les Systèmes UNIX

Michel Soto

Université Paris Descartes

Signal

Definition

Signal: interruption logicielle générée à la suite d'un évènement

Utilité

- Gestion des évènements asynchrones.

Évolution

- Implémentés dés les premières version d'Unix
  - Peu fiables: des signaux se perdent
  - Difficulté de masquer les signaux en section critique
- 4.3BSD et SVR3 modifient leurs implémentations pour fournir des signaux fiables
  - Implémentations Berkley et AT&T incompatibles !
- Normalisation POSIX.1 en 1988 et 2001
  - Fiable et portable
  - Implémenté par tous

Identification des signaux

- Chaque signal possède un nom symbolique de préfixe `SIG`
  - Définition dans `<signal.h>`
- Chaque signal possède un numéro $\in$ [1..`NSIG`]
  - La valeur 0 est réservée à un usage particulier de la fonction `kill`

A NOTER

- Le nombre de type de signaux supporté n'est pas identique partout
  - Linux: 31
  - Solaris 10: 40

  sans compter les types signaux dédiés aux applications temps réel

Propriétés

- Un événement particulier est associé à tout signal
- Le type est la seule information véhiculée par un signal (l'événement associé peut ou ne pas s'être produit)
- L'événement associé à un signal peut être soit :
  - Un événement externe au processus
    - frappe de caractère sur un terminal
    - terminaison d'un autre processus
    - envoi d'un signal par un autre processus
    - ...
  - Un événement interne au processus
    - erreur arithmétique
    - violation mémoire
    - ...

État d'un signal

Définition

Signal ***généré*** ou ***envoyé/émis***
L'événement associé au signal s'est produit

Définition

Signal ***délivré*** ou ***pris en compte***
L'action associée au signal a été exécutée

Définition

Signal ***pendant***
Le signal émis n'a pas encore été pris en compte

ATTENTION

Une seule occurrence d'un même type de signal peut être **pendante**

Définition

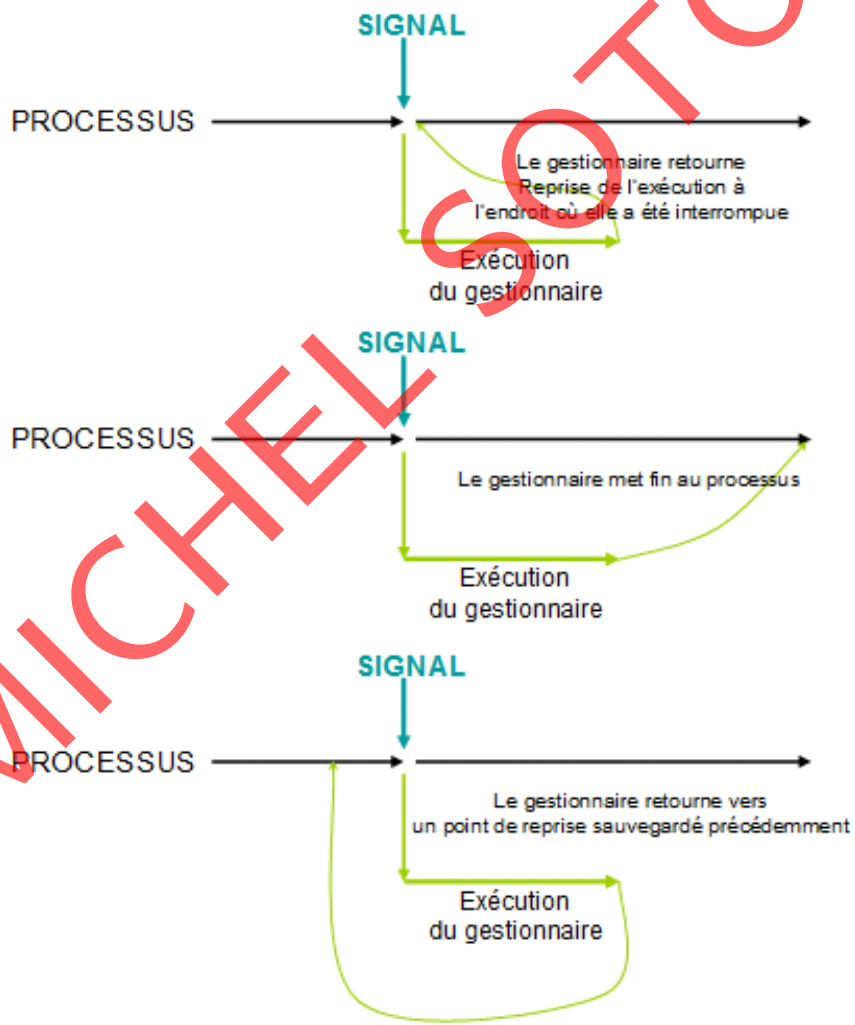
Signal ***bloqué*** ou ***masqué***
La prise en compte du signal est volontairement différée.
Le signal n'aura d'effet que lorsque le processus l'aura débloqué

Moments de délivrance d'un signal

- Un signal est délivré à son processus destinataire lorsque celui-ci :
  - vient d'être élu pour le processeur
  - revient d'un appel système
  - revient d'une interruption matérielle
- **Aucun signal n'a d'effet sur un processus zombie**

Prise en compte d'un signal

La délivrance d'un signal entraîne l'exécution d'une fonction particulière appelée *gestionnaire* (handler)



Prise en compte d'un signal (Suite)

- Gestionnaire du système *SIG_DFL*. Il exécute l'action par défaut associée au signal. Soit :
  - Le processus se termine (action la plus fréquente)
  - Le processus se termine avec image mémoire (core dumped)
  - Le signal est ignoré
  - Le processus est suspendu
  - Le processus est repris

- Gestionnaire du système *SIG_IGN*
  - L'action exécutée consiste à ... *ne rien faire !!*
  - Tout se passe comme si le signal n'avait pas été envoyé au processus
    - `SIGKILL` `SIGSTOP` et `SIGSCONT` ne peuvent jamais être ignorés
      Toute tentative est ignorée **silencieusement**

Prise en compte d'un signal (Fin)

- Gestionnaire ***fourni par l'utilisateur***
  - Dans ce cas le signal est dit *capté* ou *intercepté*
  - Le gestionnaire exécute l'action codée par le développeur

ATTENTION : signal bloqué lors de l'exécution du gestionnaire

Pendant la durée de l'exécution du gestionnaire associé à un signal, ce dernier est **bloqué** que le gestionnaire soit :

- le gestionnaire par défaut
- un gestionnaire fourni par l'utilisateur

ATTENTION : signal bloqué $\neq$ signal ignoré

- Signal bloqué : la délivrance du signal est différée.
  Le signal existe tant qu'il est bloqué ET que le système ne l'a pas délivré au le processus
- Signal ignoré : le signal est délivré, il n'existe plus

Les primitives d'émission d'un signal

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

Envoie un signal à un processus ou un groupe de processus

- `pid` : destinataire(s) du signal
  - > 0 : processus d'identité `pid`
  - $= 0$: tous les processus du même groupe ID que l'émetteur et vers lesquels l'émetteur est autorisé
  - $= -1$: tous les processus vers lesquels l'émetteur est autorisé
    Non définie par POSIX (broadcast signal, SVR4 et 4.3+BSD)
  - < -1 : tous les processus vers lesquels l'émetteur est autorisé et dont le groupe ID $= |$`pid`$|$
- `signo` : numéro du signal
  - < 0 : valeur incorrecte
  - $>$ `NSIG` : valeur incorrecte
  - $\in$ [1..`NSIG`] : signal de numéro `signo`

Retourne : 0 en cas de succès ou -1 en cas d'échec

Les primitives d'émission d'un signal (Suite)

```
#include <sys/types.h>
#include <signal.h>
int raise(int signo);
```

Envoie un signal au processus appelant

- `raise(signo)` est équivalent à `kill(getpid(), signo)`

Retourne: 0 en cas de succès ou -1 en cas d'échec

A NOTER

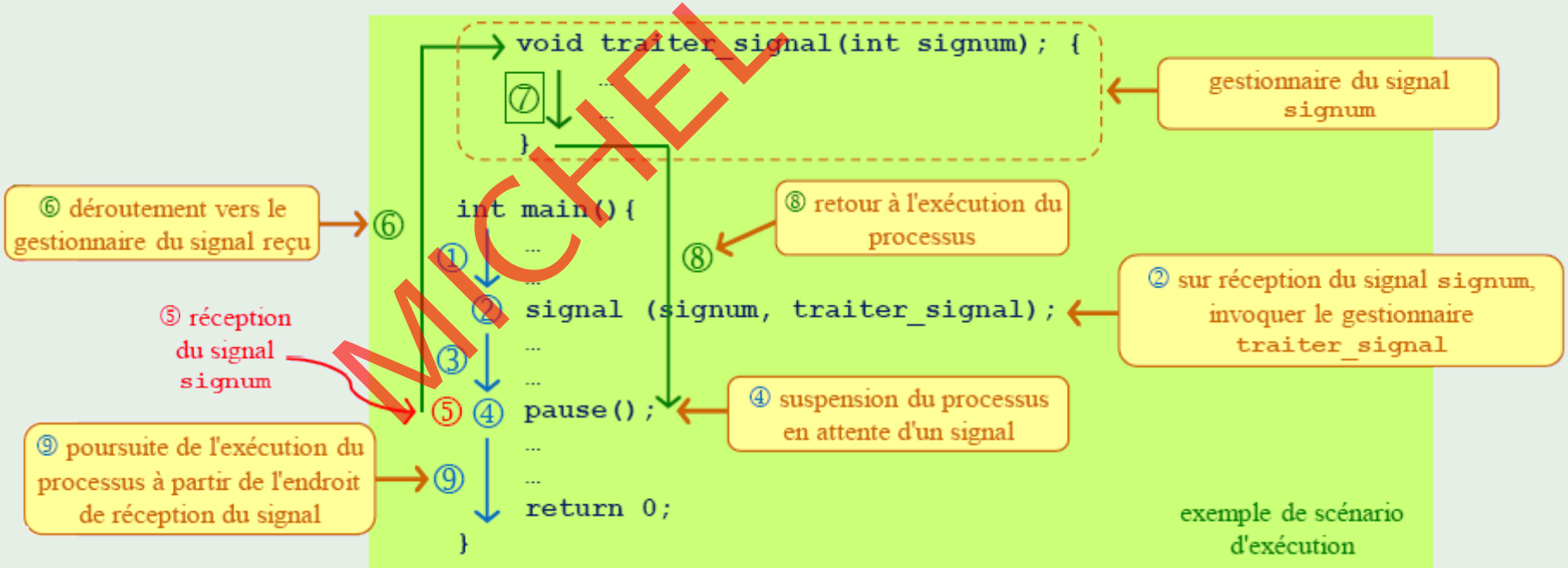
- Un processus a besoin de droit pour envoyer un signal à un autre processus
  Règle de base: le user ID réel de l'émetteur doit être égal au user ID réel ou effectif du destinataire
- Le super utilisateur peut envoyer un signal à n'importe quel processus
- POSIX.1 définit un non-signal qui à la valeur 0
  - les contrôles d'erreurs sont effectués mais aucun signal n'est envoyé
  - si `kill` retourne -1 et `errno` = `ESRCH` alors le processus destinataire n'existe pas
  - sinon il existe **ou son PID a été recyclé par le système**

Attendre un signal

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

Suspend le processus appelant jusqu'à l'arrivée d'un signal quelconque
Retourne -1 et `errno = EINTR` si un signal a été capté et que le gestionnaire du signal s'est terminé

Exemple



Installation d'un gestionnaire avec `signal`

```
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
```

Installe le gestionnaire spécifié par `func` pour le signal `signo`.

- `signo` : signal pour lequel le gestionnaire est installé
- `func` : fonction (gestionnaire) qui sera exécutée lors de la prise en compte du signal
  - Elle a pour **paramètre d'appel un** `int` qui est le numéro du signal qui déclenché l'exécution de la fonction
    Un *même gestionnaire* peut être installé pour *plusieurs signaux* (cf. exemples)
  - Elle ne retourne RIEN

Retourne : un pointeur sur la valeur antérieure du gestionnaire, ou `SIG_ERR` en cas d'échec

Installation avec `signal` d'un gestionnaire fourni par le *système*

Exemple 1

```
#include <stdio.h>
#include <signal.h>
int main(){
        printf("Avant le \"pause\" !\n");
        pause();
        printf("Après le \"pause\" !\n");
        return 0;
}

> ./signal_dfl
Avant le "pause" !
^C
>
```


Taper "Control C" génère le signal SIGINT
Le comportement par défaut associé à ce signal est: **terminaison du programme**

Installation avec `signal` d'un gestionnaire fourni par le *système*

Exemple 2

```
#include <stdio.h>
#include <signal.h>
int main(){
        signal(SIGINT, SIG_IGN); // Ignorer le signal SIGINT
        printf("Avant le \"pause\" !\n");
        pause();
        printf("Après le \"pause\" !\n");
        return 0;
}

> ./signal_ign
Avant le "pause" !
^C^C^C^C^C^C^C^C^C^C^C
```

Taper "Control C" génère le signal SIGINT
Le signal est ignoré: **le programme de réagit pas**

Installation d'un gestionnaire *utilisateur* avec `signal` (signal capté)

Exemple 3

```
#include <stdio.h>
#include <string.h>
#include <signal.h>
void traiter_signal(int signum){
     printf("Signal %s reçu !\n", strsignal(signum));
}
int main(int argc, char **argv){
        signal(SIGINT, traiter_signal);
        while(1){
                printf("Avant le \"pause\" !\n");
                pause();
                printf("Après le \"pause\" !\n");
        }
        return 0;
}
 ./signal_gest
Avant le "pause" !
^CSignal Interrupt reçu !
Après le "pause" !
Avant le "pause" !
```


1) Le signal `SIGINT` est capté
2) pause retourne
3) le programme reprend après le retour de pause

Installation d'un gestionnaire *utilisateur* avec `signal` (signal capté)

Exemple 4

```
#include <stdio.h>
#include <stdio.h>
#include <signal.h>
int compteur; int captation=0;
void traiter_signal (int sig) {
    printf ("\nGestionnaire\tCompteur:\t\t\t%d\n", compteur);
    captation=1; return;
}
main(){ signal(SIGUSR1, traiter_signal); signal(SIGUSR2, traiter_signal);
        for (;;) {compteur++;
            if (captation) {printf("Main\t\tCompteur après captation\t%d\n", compteur); captation=0;}
        }
}// main
> ./a.out &
[2] 31464
> kill -10 31464
Gestionnaire    Compteur:                   155656990
Main            Compteur après captation    155656991
> kill -12 31464
Gestionnaire    Compteur:                   75408668
Main            Compteur après captation    75408668
```


1) Le signal `SIGUSR1` ou `SIGUSR2` est capté
2) Le gestionnaire `traiter_signal` s'exécute puis retourne
3) le programme reprend sa boucle là où il à été interrompu

Installation avec `signal` d'un gestionnaire *utilisateur* (signal capté)

Exemple 5

```
#include <stdio.h>
#include <signal.h>
#include <string.h>
unsigned int s;
void traiter_signal (int sig) {printf("Reception de: %s\n", strsignal(sig)); return;
}
main(){    signal(SIGUSR1, traiter_signal);
           signal(SIGUSR2, traiter_signal);
           s=sleep (50);
           printf ("REVEIL %d SECONDES AVANT LA FIN DE MA SIESTE\n", s);
}
> ./a.out &
[1] 866
> kill -10 866
> Reception de: User defined signal 1
REVEIL 42 SECONDES AVANT LA FIN DE MA SIESTE

> ./a.out &
[1] 883
> kill -12 883
> Reception de: User defined signal 2
REVEIL 11 SECONDES AVANT LA FIN DE MA SIESTE
```

Installation d'un gestionnaire avec `signal`

ATTENTION

`signal` est l'interface historique. Son comportement varie selon les systèmes et les versions de système. Lorsque signal est capté

- sur SysV : le gestionnaire est réinitialisé à `SIG_DFL`
- sur BSD : le gestionnaire courant reste installé

Pendant l'exécution d'un gestionnaire

- sur SysV : le handler peut être interrompu pour le même signal
- sur BSD : le signal qui a provoqué l'exécution du gestionnaire est masqué

Utiliser les fonctions POSIX !

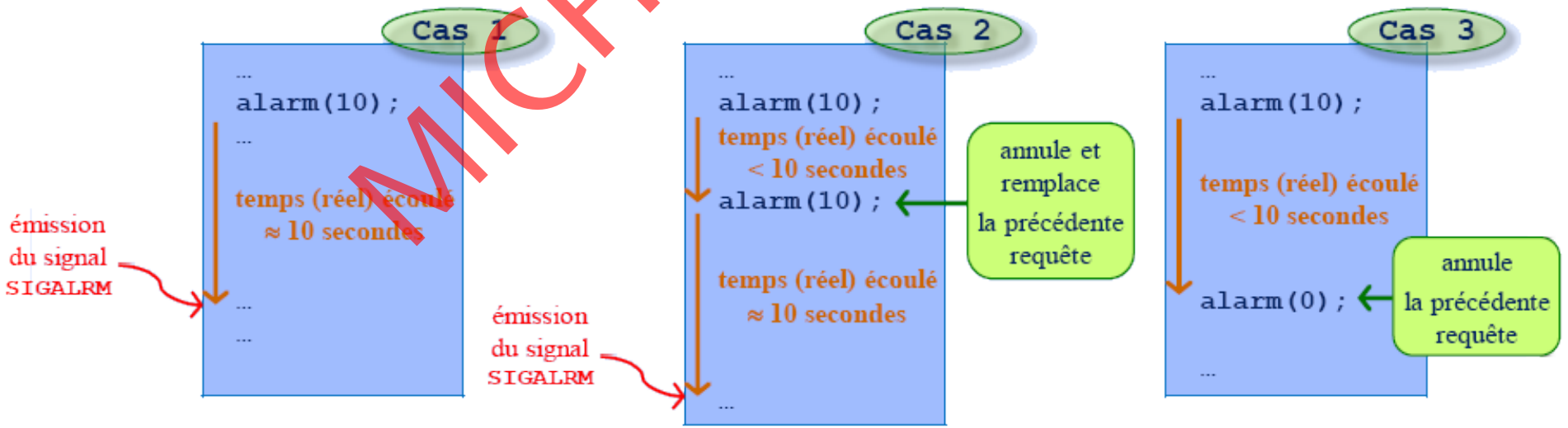
La primitive alarm

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

Demande au système d'envoyer au processus appelant le signal SIGALRM dans, au plus tôt, secondes secondes.
– Utile pour implémenter des temporisateurs

- Si secondes > 0 : annule et remplace une éventuelle requête déjà en cours
- Si secondes $= 0$: annule la précédente requête (**sans la remplacer**)

Retourne le nombre de secondes restantes avant que la précédente requête génère le signal SIGALRM ou 0 si aucune requête n'est en cours (jamais d'échec)



La primitive alarm

Exemple

```
#include <stdio.h>
#include <unistd.h>
#define DELAI 5
int main(int argc, char **argv){
    int valeur;
    alarm(DELAI); // L'utilisateur à DELAI sec. pour entrer sa valeur
                  // avant que SIGALRM ne cause la fin du processus
    printf ("Vous avez %d s pour saisir une valeur: ",DELAI);
    scanf("%d", &valeur);
    alarm(0); // Annulation de la requête alarm(DELAI)
    printf ("Vous avez saisi: %d\n", valeur);
    return 0;
}
>./signal_alarm
Vous avez 5 s pour saisir une valeur: Minuterie d'alerte
>./signal_alarm
Vous avez 5 s pour entrer une valeur: 7
Vous avez saisi: 7
```


1) L'utilisateur a tardé : la réception de SIGALRM cause la terminaison du programme
2) L'utilisateur est assez rapide : le programme lit la valeur et termine normalement

Les fonctions POSIX

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act, struct sigaction *oldact);
```

Permet de déterminer ou de modifier l'action associée à un signal particulier

- `signo` : signal pour lequel le gestionnaire est installé
- si `act` $\neq$ NULL, il s'agit d'une modification de l'action associée au signal `signo`
- si `oldact` $\neq$ NULL, le système retourne l'ancienne action pour le signal `signo`

```
struct sigaction {
    void (*sa_handler)();     // adresse du handler, ou SIG_IGN, ou SIG_DFL
    sigset_t sa_mask;         // signaux additionnels à bloquer
    int sa_flags;             // options (SA_RESTART, SA_NOCLDWAIT,
                              //          SA_NODEFER, SA_NORESETHAND, ... )
}
```

Retourne : 0 en cas de succès et -1 sinon

Ensembles de signaux

- Utilisés pour la manipulation de plusieurs signaux

```
#include <signal.h>
int sigemptyset(sigset_t *set);
```

Initialise à VIDE l'ensemble de signaux pointé par `set`
Retourne 0 en cas de succès ou -1 en cas d'erreur

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

Initialise l'ensemble de signaux pointé par `set` avec TOUS les signaux existant sur la machine
Retourne 0 en cas de succès ou -1 en cas d'erreur

Ensembles de signaux (Suite)

```
int sigaddset(sigset_t *set, int signo);
```

Ajoute le signal `signo` à l'ensemble de signaux pointé par `set`
Retourne 0 en cas de succès ou -1 en cas d'erreur

```
int sigdelset(sigset_t *set, int signo);
```

Supprime le signal `signo` de l'ensemble de signaux pointé par `set`
Retourne 0 en cas de succès ou -1 en cas d'erreur

```
int sigismember(const sigset_t *set, int signo);
```

Teste l'appartenance du signal `signo` à l'ensemble de signaux pointé par `set`
Retourne 1 si `signo` appartient à l'ensemble, 0 si `signo` n'appartient pas à l'ensemble
ou -1 en cas d'erreur

Masque des signaux d'un processus

Définition

Ensemble courant des signaux bloqués par le processus

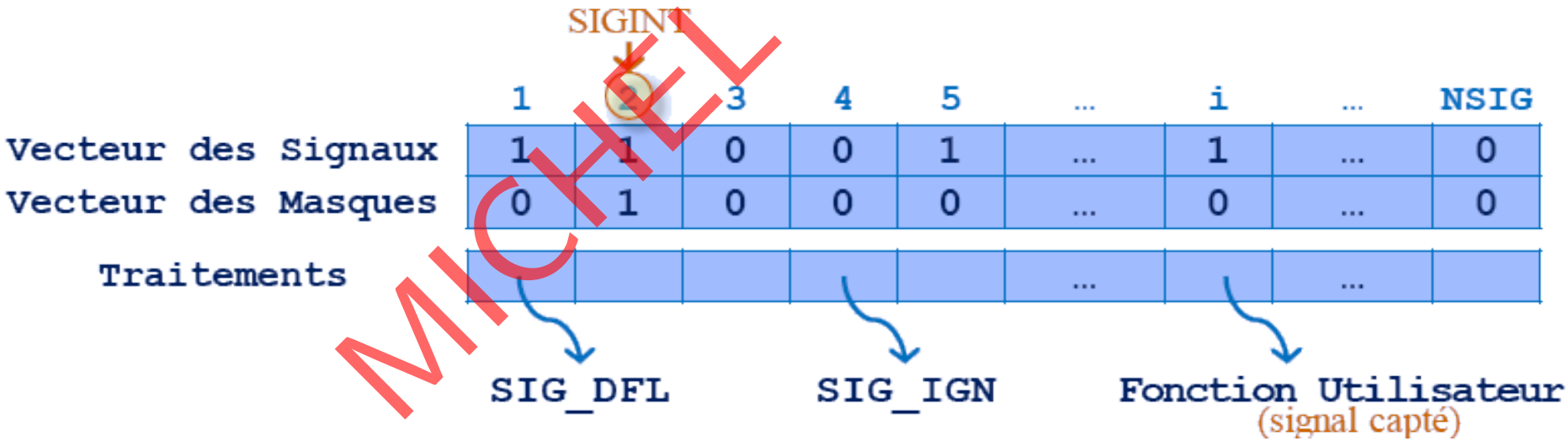
ATTENTION

- Si plusieurs occurrences d'un même signal bloqué sont générées :
  - POSIX.1 autorise le système à délivrer une ou plusieurs occurrences de signal
    - Quand le système délivre plusieurs plusieurs occurrences de signal, celles-ci sont mise en file d'attente pour le processus destinataire
  - La plus part des systèmes Unix ne délivrent qu'**une seule occurence** du signal.
    - Si un signal S est déjà pendant alors les autres occurences de S sont **perdues** (elles ne sont pas mémorisées) sauf si l'*extension temps réel* de POSIX.1 est prise en charge

Masque des signaux d'un processus (Suite)

Implémentation

- Pour chaque processus, le système gère un vecteur de bit
- Un seul bit est associé à chaque signal pris en charge par le système
  - Bit à 0 : le signal n'est pas bloqué
  - Bit à 1 : le signal est bloqué



Consultation, modification du masque des signaux d'un processus

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oldset);
```

- Si `oldset` $\neq$ NULL, le masque courant du processus est retourné à travers `oldset`.
- Si `set` $\neq$ NULL alors `how` indique comment le masque courant est modifié :
  - `SIG_BLOCK` $\rightarrow$ *nouveau_masque* $= \{$*masque_courant* $\cup$ `set`$\}$
  - `SIG_UNBLOCK` $\rightarrow$ *nouveau_masque* $= \{$*masque_courant* $-$ `set`$\}$
  - `SIG_SETMASK` $\rightarrow$ *nouveau_masque* $= \{$`set`$\}$
- Si `set` $=$ NULL alors le masque courant n'est pas modifié et `how` est ignoré

Retourne 0 en cas de succès et -1 sinon

ATTENTION !!

- S'il existe des **signaux pendants non-bloqués** après l'appel à `sigprocmask` alors **au moins un** de ces signaux **est délivré** au processus **avant le retour de la primitive**
- `sigprocmask` ne doit pas être utilisé pour les processus multi-threadés (`pthread_sigmask`)

Signaux pendants

```
#include <signal.h>
int sigpending(sigset_t *set);
```

Permet de connaître les signaux pendants du processus appelant

- `set` : ensemble des signaux pendants (c.-à-d. bloqués et non délivrés) du processus

Retourne 0 en cas de succès et -1 sinon

Affichage des masques de signaux d'un processus

Exemple

```
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
...
/* ================================================ */
   void affich_mask (char *message, sigset_t *mask){
/* ================================================ */
sigset_t set_signaux;
int i;
if (mask == NULL){// Affichage du masque courant
    if (sigprocmask(0, NULL, &set_signaux)<0) {
        perror("pb sigprocmask"); exit (EXIT_FAILURE);
    }
}
else set_signaux= *mask; // Affichage du masque passé en paramètre

printf("%s{", message);
for (i=1; i< NSIG; i++){
    if (sigismember(&set_signaux, i)) printf("%d ", i);
}
printf("}\n");
}// affich_mask
```

Attente d'un signal avec remplacement temporaire du masque des signaux

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

- Remplace temporairement le masque courant des signaux bloqués par `sigmask`
- Met en sommeil (suspend) le processus jusqu'à l'arrivée soit :
  - d'un signal non masqué, non ignoré et non capté qui met fin au processus
  - d'un signal non masqué, non ignoré et capté.
    Si le gestionnaire se termine
    alors `sigsuspend` :
    — **se termine**,
    — **retourne au processus appelant**
    — le masque des signaux est restauré à sa valeur avant l'appel à `sigsuspend`
    sinon `sigsuspend` ne retourne pas et le processus se termine

Retourne **toujours** -1 et `errno` = `EINTR` (appel système interrompu)

IMPORTANT

- Remplacement du masque et mise en sommeil du processus sont réalisés de manière **atomique**

Installation d'un gestionnaire avec `sigaction`

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act, struct sigaction *oldact);
```

Permet de déterminer ou de modifier le gestionnaire associé à un signal particulier.

- `signo` : signal concerné
- Si `act` $\neq$ NULL : il s'agit d'une modification du gestionnaire associé au signal `signo`.
- Si `oldact` $\neq$ NULL : retourne le gestionnaire actuel associé au signal `signo`

Retourne 0 en cas de succès et -1 sinon

IMPORTANT

- Un gestionnaire pour un signal donné demeure installé tant qu'il n'est pas remplacé lors d'un autre appel à `sigaction`

Installation d'un gestionnaire avec `sigaction` (Suite)

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act, struct sigaction *oldact);
struct sigaction {
                void (*sa_handler)();
                sigset_t sa_mask;
                int sa_flags;
}
```

- `sa_handler` : gestionnaire utilisateur ou `SIG_IGN` ou `SIG_DFL`
- `sa_mask` : signaux additionnels à bloquer **au cours de l'exécution du gestionnaire**
  Si le gestionnaire retourne, le masque des signaux est restauré à sa valeur précédente
  - `signo` est **toujours masqué**
  - d'autres signaux peuvent être ajouter en plus de `signo` (avec `sigemptyset`, `sigaddset`, etc)
- `sa_flags` : options de traitement du signal
  Usage : `act.sa_flags = flag1 | flag2 | ...;`

Quelques options de traitement des signaux avec `sigaction`

- `SA_NOCLDWAIT` : les fils qui se terminent ne deviennent pas zombies
  - N'a de sens que si le gestionnaire est associé à `SIGCHLD`
  - POSIX.1 ne spécifie pas si `SIGCHLD` est généré lorsqu'un processus fils se termine.
    - Sous Linux, un signal `SIGCHLD` est généré dans ce cas
    - Sur d'autres implémentations, il ne l'est pas
- `SA_NODEFER` : autorise la réception d'un signal **au cours de l'exécution du gestionnaire installé pour ce même signal**
- `SA_RESETHAND` : réinstalle le gestionnaire `SIG_DFL` après l'appel du gestionnaire actuel
  - Interdit pour `SIGILL` et `SIGTRAP`
- `SA_RESTART` : si un appel système a été interrompu par le signal, il est redémarré automatiquement
  - **Par défaut**, tout appel système bloquant interrompu par un signal **échoue avec `errno` = `EINTR`**

Quelques options de traitement des signaux avec sigaction (Suite)

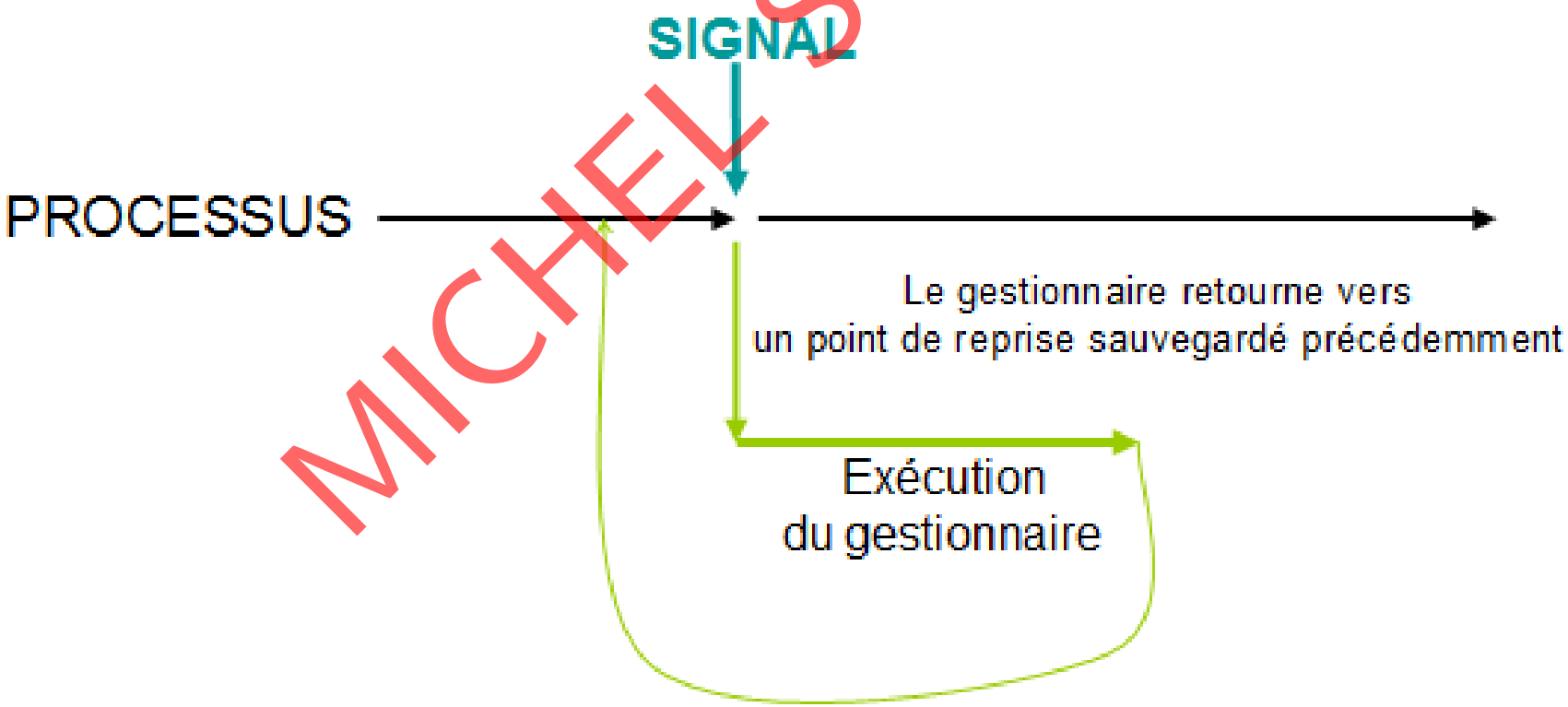
- SA_SIGINFO : le prototype du gestionnaire n'est plus
  `void handler(int signo);`
  mais
  `void handler(int signo, siginfo_t info, void context);`
  - La structure siginfo renseigne sur les causes qui ont conduit à la génération du signal reçu

```
struct siginfo {
int si_signo;  /* numéro du signal */
int si_errno;  /* si différent de zero, valeur de errno  */
int si_code;   /* info additionnelle selon le signal) */
pid_t si_pid;  /* PID du processus émetteur */
uid_t si_uid;  /* UID réel de l'émétteur */
void *si_addr; /* addresse qui a causé l'erreur */
int si_status; /* valeur du exit ou numéro du signal  */
union sigval si_value; /* valeur propre à l'application */
/* D'autres champs sont possibles selon les implémentations */
};
```

Contrôle du point de reprise

La gestion d'erreurs de bas niveau peut être plus efficace en retournant dans la boucle principale du programme plutôt qu'après l'endroit du programme où l'erreur s'est produite.
Pour cela, il faut pouvoir :

- mémoriser le point de reprise
- retourner vers le point de reprise mémorisé



Contrôle du point de reprise (Suite)

```
#include <setjmp.h>
                int sigsetjmp(sigjmp_buf env, int savemask);
```

Mémorise l'environnement et le masque courant

- `env` : variable où sera mémorisé le contexte d'exécution du point de reprise
- `savemask` : si $\neq 0$, le masque courant des signaux est aussi sauvegardé dans `env`

Retourne 0 pour un appel direct, et une valeur non nulle si elle retourne depuis un appel à `siglongjmp()`.

```
#include <setjmp.h>
                void siglongjmp(sigjmp_buf env, int val);
```

Simule un retour de l'appel à la fonction `sigsetjmp()` avec un retour de valeur égal à `val` (si `val` $= 0$, alors retour de la valeur 1),
Restaure l'environnement sauvegardé par l'appel à `sigsetjmp()`.
Si le masque des signaux a été sauvegardé par `sigsetjmp`, il est aussi restauré.

- `env` : variable où a été mémorisé le contexte d'exécution du point de reprise
- `val` : valeur qui sera retournée par le retour simulé de `sigsetjmp`

Cette fonction ne retourne jamais

Contrôle du point de reprise (Suite)

Exemple 1/2

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

sigjmp_buf env; // Pour la mémorisation du point de reprise
struct sigaction action;

void signal_FPE (int signal) {printf ("Réception de SIGFPE\n");
                              siglongjmp(env, 1);// Retourne au point de reprise
} // signal_FPE

int fonctionC (int a, int b) { return (a/b);
} // fonctionC

int fonctionB (int a, int b) {
    int r=fonctionC(a, b); printf ("fonctionC retourne %d\n", r);
    return (r);
} // fonctionB

int fonctionA (int  a, int b) {
    int r=fonctionB(a, b); printf ("fonctionB retourne %d\n", r);
    return (r);
} // fonctionA
```

Exemple 2/2

```
int main(int argc, char **argv)
{   int s; int resultat;
    action.sa_handler=signal_FPE;
    sigaction (SIGFPE, &action, NULL); // Installation de signal_FPE

    s=sigsetjmp (env, 1); // Mémorisation du point de reprise

    if (s==0) {printf ("Retour direct de sigsetjmp\n");
               resultat=fonctionA(atoi(argv[1]), atoi (argv[2]));
               printf ("fonctionA retourne %d\n", resultat);
    }
    else {printf ("Retour de sigsetjmp par siglongjmp\n");
          printf ("Division par 0 dans la fonction C\n");
    }
} // main
```

```
 > ./a.out 4 2
Retour direct de sigsetjmp
fonctionC retourne 2
fonctionB retourne 2
fonctionA retourne 2
```

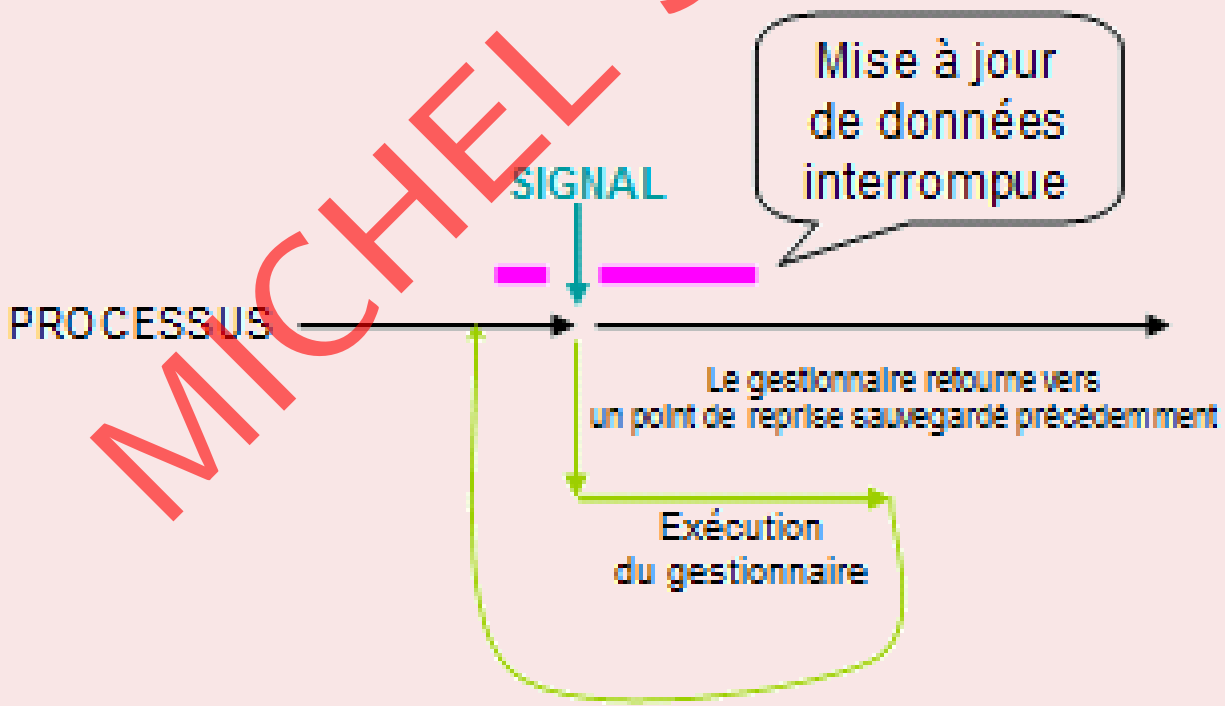
```
 > ./a.out 6 0
Retour direct de sigsetjmp
Réception de SIGFPE
Retour de sigsetjmp par siglongjmp
Division par 0 dans la fonction C
```

Contrôle du point de reprise (Fin)

IMPORTANT

L'usage de `sigsetjmp/siglongjmp` doit être raisonné car :

- son usage intensif rend difficile la compréhension de la structure du programme (structure *spaghettis*)
- il peut conduire à des mises à jour incomplètes de structures de données
  - Le processus ne reprend pas son exécution à l'endroit où la mise à jour a été interrompue



Reprise des appels systèmes interrompus suite à un signal

- Dans les premiers systèmes UNIX, un appel système *lent* était interrompu
  - L'appel retourne une erreur et `errno` = `EINTR`

- Appel système *lent* : appel système qui peut être bloquant indéfiniment
  - Lecture/Ecriture sur certains types de fichiers
    - Pipe
    - Terminal
    - Réseau
  - Primitive `pause`
    - Bloque un processus jusqu'à l'arrivée d'un signal
  - Primitive `wait`
    - Bloque un processus jusqu'à la fin d'un de ses fils
  - Certaines opérations `ioctl`
  - Certaines fonction de communication inter processus

Les disques ne sont pas considérés comme des périphériques *lents*

Reprise des appels systèmes interrompus suite à un signal

Problème

- Il faut maintenant se préoccuper des appels système interrompus

```
for (n=read (fd, buf, BUFFSIZE); errno==EINTR; n=read (fd, buf, BUFFSIZE));
if (n<0){
    perror ("PB avec read);
    exit (EXITFAILURE);
}
```

  - Cela alourdi le code
  - Pour l'utilisateur, certains signaux peuvent ne présenter aucun intérêt (`SIGCHLD`)
  - L'utilisateur ne sait pas toujours s'il utilise un périphérique *lent*

Solution

- Redémarrage automatique des appels système interrompus
  - 4.3BSD, 4.4BSD, FreeBSD, Linux, Mac OS X
    - Primitive `signal` : reprise automatique par défaut
    - Primitive `sigaction` : optionnel

Reprise des appels systèmes interrompus suite à un signal

Exemple *sans* SA_RESTART (1/2)

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

void traiter_signal(int signum){
        printf("\t\tDébut gestionnaire !\n");
        sleep(5);
        printf("\t\tFin gestionnaire !\n");
}

int main(int argc, char **argv){
    struct sigaction fonc;
    char buf[128]; int r;
    fonc.sa_handler = traiter_signal;

    if ( sigaction(SIGINT, &fonc, NULL) == -1 ) perror("Erreur sigaction !\n");
    else printf("Installation du gestionnaire pour SIGINT\n");

    printf("\tSaisir une valeur\n");
    r=read(STDIN_FILENO, buf, sizeof(buf));
    if (r<0 && errno==EINTR) perror("\tErreur read");
    else printf ("\tValeur lue: %s\n",buf);

    return 0;
}
```

Exemple *sans* SA_RESTART (2/2)

```
> ./sigaction_sa_restart_no
Installation du gestionnaire pour SIGINT
        Saisir une valeur
^C              Début gestionnaire !
coucou
                Fin gestionnaire !
        Erreur read: Interrupted system call
> coucou
> coucou : commande introuvable
>
```


- La frappe de "Control C" interromp l'appel système **read qui n'est pas redémarré** après l'exécution du gestionnaire
- La valeur "coucou" **n'est pas lue par le programme** mais par ... le shell

Exemple *avec* SA_RESTART (1/2)

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

void traiter_signal(int signum){
        printf("\t\tDébut gestionnaire !\n");
        sleep(5);
        printf("\t\tFin gestionnaire !\n");
}
int main(int argc, char **argv){
    struct sigaction fonc;
    char buf[128]; int r;
    fonc.sa_handler = traiter_signal;
/**/fonc.sa_flags = SA_RESTART;

    if ( sigaction(SIGINT, &fonc, NULL) == -1 ) perror("Erreur sigaction !\n");
    else printf("Installation du gestionnaire pour SIGINT\n");

    printf("\tSaisir une valeur\n");
    r=read(STDIN_FILENO, buf, sizeof(buf));

    if (r<0 && errno==EINTR) perror("\tErreur read");
    else printf ("\tValeur lue: %s\n",buf);
    return 0;
}
```

Exemple *avec* SA_RESTART (2/2)

```
> ./sigaction_sa_restart_yes
Installation du gestionnaire pour SIGINT
        Saisir une valeur
^C              Début gestionnaire !
coucou
                Fin gestionnaire !
        Valeur lue: coucou
>
```

- La frappe de "Control C" interromp l'appel système **read** **qui est redémarré automatiquement** après l'exécution du gestionnaire
- La valeur "coucou" est **correctement lue par le programme**

Signal non masqué lors de l'exécution de son gestionnaire

Exemple *avec* SA_NODEFER (1/2)

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>
int a;
void traiter_signal(int signum){
    int appel;
    appel=++a;
    printf("\t\tDébut gestionnaire ! (%d)\n", appel);
    sleep(5);
    printf("\t\tFin gestionnaire ! (%d)\n", appel);
}
int main(int argc, char **argv){
    struct sigaction fonc;
    fonc.sa_handler = traiter_signal;
/**/fonc.sa_flags = SA_NODEFER;

    if ( sigaction(SIGINT, &fonc, NULL) == -1 ) perror("Erreur sigaction !\n");
    else printf("Installation du gestionnaire pour SIGINT\n");

    while(1){ a=0;
              printf("\tAvant le \"pause\" !\n");
              pause();
              printf("\tAprès le \"pause\" !\n");
    }
    return 0;
}
```

Signal non masqué lors de l'exécution de son gestionnaire

Exemple *avec* SA_NODEFER (2/2)

```
> ./sigaction_sa_nodefer
Installation du gestionnaire pour SIGINT
        Avant le "pause" !
^C              Début gestionnaire ! (1)
^C              Début gestionnaire ! (2)
^C              Début gestionnaire ! (3)
                Fin gestionnaire ! (3)
                Fin gestionnaire ! (2)
                Fin gestionnaire ! (1)
        Après le "pause" !
        Avant le "pause" !
>
```


- Avec SA_NODEFER, **le signal SIGINT n'est pas masqué** pendant l'exécution de son gestionnaire
- Chaque frappe de "Control C" provoque un nouvel appel au gestionnaire qui s'empile sur les précédents appels. Ici 3 appels sont empilés.

Restauration du gestionnaire par défaut après l'exécution du gestionnaire d'un signal capté

Exemple *avec* SA_RESETHAND (1/2)

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

void traiter_signal(int signum){
        printf("\t\tDébut gestionnaire !\n");
        printf("\t\tFin gestionnaire !\n");
}
int main(int argc, char **argv){
    struct sigaction fonc;
    fonc.sa_handler = traiter_signal;
/**/fonc.sa_flags = SA_RESETHAND;

    if ( sigaction(SIGINT, &fonc, NULL) == -1 ) perror("Erreur sigaction !\n");
    else printf("Installation du gestionnaire pour SIGINT\n");

    while(1){
                printf("\tAvant le \"pause\" !\n");
                pause();
                printf("\tAprès le \"pause\" !\n");
    }
    return 0;
}
```

Restauration du gestionnaire par défaut après l'exécution du gestionnaire d'un signal capté

Exemple *avec* SA_RESETHAND (2/2)

```
> ./sigaction_sa_resethand
Installation du gestionnaire pour SIGINT
        Avant le "pause" !
^C              Début gestionnaire !
                Fin gestionnaire !
        Après le "pause" !
        Avant le "pause" !
^C
>
```


- Lors du premier "Control C", le signal SIGINT est capté par le gestionnaire et le processus continue son exécution (boucle)
- Lorsque le gestionnaire retourne, l'option SA_RESETHAND provoque la restauration du gestionnaire par défaut SIGDFL
- Lors du second "Control C", le signal SIGINT n'est plus capté par le gestionnaire. L'action par défaut associée à s'exécute: terminaison du processus

Exemple : utilisation conjointe de sigprocmask et de sigsuspend (1/2)

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

void traiter_signal(int signum){
  affich_mask("Masque dans traiter_signal ", NULL);
  return;
}

int main(int argc, char **argv){
sigset_t s_bloques, s_courant,
         s_attente, s_pending;
struct sigaction action;

// Installation du gestionnaire pour SIGINT
   action.sa_handler=traiter_signal;
   sigemptyset(&action.sa_mask);
   if (sigaction (SIGINT, &action, NULL)<0)
      {perror("!!!pb signal"); exit (EXIT_FAILURE);}

   affich_mask("Masque au début du programme ", NULL);

// Ajout de SIGINT dans s_bloques
   sigemptyset (&s_bloques);
   sigaddset (&s_bloques, SIGINT);

// Ajout de SIGUSR1 dans s_attente
   sigemptyset (&s_attente);
   sigaddset (&s_attente, SIGUSR1);
```

```
 // Blocage de SIGINT et sauvegarde du masque courant
/*1*/sigprocmask(SIG_SETMASK, &s_bloques, &s_courant);

     affich_mask("Masque en section critique ", NULL);

     sleep (7);// Entrer Control C pendant ce temps

     sigpending (&s_pending);
     affich_mask("Signaux pendants en section critique ",
                 &s_pending);

     printf ("ATTENTE...\n");

  // Autorisation de tous les signaux sauf SIGUSR1
     sigsuspend (&s_attente);

  // ICI: Au retour de sigsuspend le masque s_bloques
  // est restauré: SIGINT est toujours bloqué
     affich_mask("Masque au retour de sigsuspend : ", NULL);

  // Déblocage de SIGINT
/*2*/sigprocmask(SIG_SETMASK, &s_courant, NULL);
     affich_mask("Masque à la fin du programme ", NULL);

     return 0;
    } // Main
```

Exemple : utilisation conjointe de sigprocmask et de sigsuspend (2/2)

```
>  ./sigpromask_section_critique
Masque au début du programme {}
Masque en section critique {2 }
^CSignaux pendants en section critique {2 }
ATTENTE...
Masque dans traiter_signal {2 10 }
Masque au retour de sigsuspend : {2 }
Masque à la fin du programme {}
>
```


- Au début de l'exécution du programme, le masque courant des signaux est *vide* ({})
- Le premier sigpromask bloque le signal SIGINT qui figure maintenant dans le masque ({2 }). Il sauvegarde aussi le masque courant
- La frappe "Control C" ne provoque pas l'exécution du gestionnaire car le signal SIGINT est bloqué.
  Il est toutefois ajouté dans le masque des signaux pendants ({2 }).

- sigsuspend change le masque courant par un masque où tous les signaux sont autorisés sauf SIGUSR1.
- Le signal SIGINT qui n'est plus bloqué est alors capté par le gestionnaire. Le signal SIGUSR1 reste bloqué ({2 10 })
- Le retour du gestionnaire provoque le retour sigsuspend qui restaure le masque tel qu'il était ({2 })
- Le second sigpromask restaure le masque initial ({})

Exemple : signal raté avec pause (1/2)

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

void traiter_signal(int signum){
  affich_mask("Masque dans traiter_signal ", NULL);
  return;
}

int main(int argc, char **argv){
sigset_t s_bloques, s_courant,
         s_attente, s_pending;
struct sigaction action;

// Installation du gestionnaire pour SIGINT
   action.sa_handler=traiter_signal;
   sigemptyset(&action.sa_mask);
   if (sigaction (SIGINT, &action, NULL)<0)
      {perror("!!!pb signal"); exit (EXIT_FAILURE);}

   affich_mask("Masque au début du programme ", NULL);

// Ajout de SIGINT dans s_bloques
   sigemptyset (&s_bloques);
   sigaddset (&s_bloques, SIGINT);

// Ajout de SIGUSR1 dans s_attente
   sigemptyset (&s_attente);
   sigaddset (&s_attente, SIGUSR1);
```

```
// Blocage de SIGINT et sauvegarde du masque courant
/*1*/sigprocmask(SIG_SETMASK, &s_bloques, &s_courant);

    affich_mask("Masque en section critique ", NULL);

    sleep (7);// Entrer Control C pendant ce temps

    sigpending (&s_pending);
    affich_mask("Signaux pendants en section critique ", &s_pendin

    printf ("ATTENTE...\n");

// Autorisation de tous les signaux sauf SIGUSR1
/*2*/sigprocmask(SIG_SETMASK, &s_attente, NULL);
    sigpending (&s_pending);
    affich_mask("Signaux pendants AVANT PAUSE ", &s_pending);

    pause ();

    affich_mask("Masque au retour de pause : ", NULL);

 // Deblocage de SIGINT
/*3*/sigprocmask(SIG_SETMASK, &s_courant, NULL);
    affich_mask("Masque à la fin du programme ", NULL);

    return 0;
    } // Main
```

Exemple : signal raté avec pause (2/2)

```
> ./sigpromask_pause
Masque au dÃ©but du programme {}
Masque en section critique {2 }
^C
Signaux pendants en section critique {2 }
ATTENTE...
Masque dans traiter_signal {2 10 }
Signaux pendants AVANT PAUSE {}
```


- Au début de l'exécution du programme, le masque courant des signaux est *vide* ({})
- Le premier `sigpromask` bloque le signal `SIGINT` qui figure maintenant dans le masque ({2 }). Il sauvegarde aussi le masque courant
- La frappe "`Control C`" ne provoque pas l'exécution du gestionnaire car le signal `SIGINT` est bloqué.
  Il est toutefois ajouté dans le masque des signaux pendants ({2 }).

- Le second `sigpromask` change le masque courant par un masque où tous les signaux sont autorisés sauf `SIGUSR1`.
- Le signal `SIGINT` qui n'est plus bloqué est alors capté par le gestionnaire. Le signal `SIGUSR1` reste bloqué ({2 10 })
- **Le programme est bloqué sur `pause` car le signal `SIGINT` à déjà était capté**
- **Il faudra un second signal `SIGINT` pour débloqué le programme**