Chapitre 11

Les signaux

Les signaux sont un mécanisme asynchrone de communication inter-processus.
Intuitivement, il sont comparables à des sonneries, les differentes sonneries indiquant des évènements différents. Les signaux sont envoyés à un ou plusieurs processus. Ce signal est en général associé à un évènement.

Peu portables entre BSD et ATT, ils deviennent plus commodes à utiliser et portables avec la norme POSIX qui utilise la notion utile de vecteur de signaux et qui fournit un mécanisme de masquage automatique pendant les procédures de traitement (comme BSD).

Un signal est envoyé à un processus en utilisant l'appel système :

```
kill(int pid, int signal);
```

`signal` est un numéro compris entre 1 et `NSIG` (défini dans `<signal.h>`) et `pid` le numéro du processus.

Le processus visé reçoit le signal sous forme d'un drapeau positionné dans son bloc de contrôle. Le processus est interrompu et réalise éventuellement un traitement de ce signal.

On peut considérer les signaux comme des interruptions logicielles, ils interrompent le flot normal d'un processus mais ne sont pas traités de façon synchrone comme les interruptions matérielles.

11.0.4 Provenance des signaux

Certains signaux peuvent être lancés à partir d'un terminal grâce aux caractères spéciaux comme `intr`, `quit` dont la frappe est transformée en l'envoi des signaux `SIGINT` et `SIGQUIT`. D'autres sont dûs à des causes internes au processus, par exemple : `SIGSEGV` qui est envoyé en cas d'erreur d'adressage, `SIGFPE` division par zéro (Floating Point Exception).

Enfin certains sont dûs à des évènements comme la déconnection de la ligne (le terminal) utilisé : si le processus leader d'un groupe de processus est déconnecté, il envoie à l'ensemble des processus de son groupe le signal `SIGHUP` (Hangup = raccrocher).

11.0.5 Gestion interne des signaux

C'est dans le bloc de contrôle (BCP) de chaque processus que l'on trouve la table de gestion des signaux (attention, sous System V < V.4, la table de gestion des processus est dans la `zone u`, c'est à dire dans l'espace-mémoire du processus).

Cette table contient, pour chaque signal défini sur la machine, une structure `sigvec` suivante :

```
{
    bit pendant;
```

```
    void (*traitement)(int);
}
```

En BSD et POSIX, on a un champ supplémentaire : `bit masque ;`
Le drapeau `pendant` indique que le processus a reçu un signal, mais n'a pas encore eu l'occasion de prendre en compte ce signal.

Remarque : comme `pendant` est un unique bit, si un processus reçoit plusieurs fois le même signal avant de le prendre en compte, alors il n'y a pas mémorisation des réceptions successives, un seul traitement sera donc réalisé.

Comme nous l'avons vu dans le graphe d'état des processus, la prise en compte des signaux se fait au passage de l'état actif noyau à l'état actif utilisateur. Pourquoi la prise en compte de signaux se fait-elle uniquement à ce moment là ?

Parce que
Une sauvegarde de la pile utilisateur et du contexte a été effectuée quand le processus est passé en mode noyau. Il n'est pas nécessaire de faire un nouveau changement de contexte. Il est facile pour traiter le signal de réaliser immédiatement une nouvelle augmentation de pile pour le traitement du signal, de plus la pile noyau est vide (remarque : en POSIX, il devient possible de créer une pile spéciale pour les fonctions de traitement de signaux).

L'appel à la fonction de traitement est réalisé de façon à ce qu'au retour de la fonction, le processus continue son exécution normalement en poursuivant ce qui était en cours de réalisation avant la réception du signal. Si l'on veut que le processus se poursuive dans un autre contexte (de pile), il doit gérer lui-même la restauration de ce contexte.

La primitive `longjmp` peut permettre de réaliser des changements de contexte interne au processus, grâce à un désempilement brutal.

Pendant ce changement d'état, la table de gestion des signaux du processus est testée pour la présence d'un signal reçu mais non traité (c'est un simple vecteur de bit pour le bit pendant, et donc testable en une seule instruction, ceci doit être fait rapidement comme le test de réception d'un signal est souvent réalisé).
Si un signal a été reçu (et qu'il n'est pas masqué), alors la fonction de traitement associée est réalisée. Le masquage permet au processus de temporiser la mise en øeuvre du traitement.

11.0.6 L'envoi de signaux : la primitive kill

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

Il y a NSIG signaux sur une machine, déclarés dans le fichier /usr/include/signal.h.
La valeur de `pid` indique le `PID` du processus auquel le signal est envoyé.

0 Tous les processus du **groupe** du processus réalisant l'appel `kill`

1 En système V.4 tous les processus du système sauf 0 et 1

pid positif le processus du pid indiqué

pid négatif tous les processus du groupe | pid |

le paramètre `sig` est interprété comme un signal si `sig` $\in$ [0-NSIG], ou comme une demande d'information si `sig` $= 0$ (suis-je autorisé à envoyer un signal à ce(s) processus ?). Comme un paramètre erroné sinon.

La fonction `raise(int signal)` est un raccourci pour `kill(getpid(), signal)`, le processus s'envoie à lui-même un signal.

Remarquez que l'on peut réécrire `kill(0, signal)` par `kill(-getpid(), signal)`. Rappel : les PID sont toujours positifs.

11.1 La gestion simplifiée avec la fonction signal

ZZZ : cette section est historique, utiliser la norme POSIX décrite plus loin.

```
ancien C : (*signal(sig, func))()
           int sig;
           int (*func)();

ANSI C :   void (*signal(int sig, void (*action)(int)))(int);
```

La fonction `signal` permet de spécifier ou de connaître le comportement du processus à la réception d'un signal donné, il faut donner en paramètre à la fonction le numéro du signal `sig` que l'on veut détourner et la fonction de traitement `action` à réaliser à la réception du signal.

Trois possibilités pour ce paramètre `action`

SIG_DFL Comportement par défaut, plusieurs possibilités
- **exit** Le processus se termine (avec si possible la réalisation d'un core)
- **ignore** Le processus ignore le signal
- **pause** Suspension du processus
- **continue** Reprise du processus si il était suspendu.

SIG_IGN le signal est ignoré.
Remarque : les signaux SIGKILL, SIGSTOP ne peuvent pas être ignorés.

HANDLER Une fonction de votre cru.

11.1.1 Un exemple

Exemple pour rendre un programme insensible à la frappe du caractère de contrôle intr sur le terminal de contrôle du processus.

```
void got_the_blody_signal(int n) {
    signal(SIGINT, got_the_blody_signal);
    printf(" gotcha!!  your  (%d) signal is useless \n");
}

main() {
    signal(SIGINT, got_the_blody_signal);
    printf(" kill me now !! \n");
    for(;;);
}
```

une version plus élégante et plus fiable :

```
signal(SIGINT, SIG_IGN);
```

11.2 Problèmes de la gestion de signaux ATT

Les phénomènes suivants sont décrits comme des problèmes mais la norme POSIX permet d'en conserver certains, mais fournit aussi les moyens de les éviter.

1. un signal est repositionné à sa valeur par défaut au début de son traitement (handler).

```
#include <signal.h>

traitement()  {
    printf("PID %d en a capture un \n", getpid());
```

```
->     reception du deuxieme signal, realisation d'un exit
    signal(SIGINT, traitement);
}

main() {
    int ppid;
    signal(SIGINT,traitement);
    if (fork()==0)
    {/* attendre que pere ait realise son nice() */
        sleep(5);
        ppid = getppid(); /* numero de pere */
        for(;;)
            if (kill(ppid,SIGINT) == -1)
                exit();
    }
/* pere ralenti pour un conflit plus sur */
    nice(10);
    for(;;) pause();  <- reception du premier signal
/* pause c'est mieux qu'une attente active */
}
```

Si l'on cherche à corriger ce défaut, on repositionne la fonction `traitement` au début du traitement du signal. Ceci risque de nous placer dans une situation de dépassement de pile : en effet, dans le programme précédent, nous pouvons imaginer que le père peut recevoir un nombre de signaux arbitrairement grand pendant le traitement d'un seul signal, d'où une explosion assurée de la pile (il suffit en effet que chaque empilement de la fonction `traitement` soit interrompu par un signal)

```
traitement(){
    signal(SIGINT,traitement);
->   signal SIGINT
    printf("PID %d en a capture un \n",getpid());
}
```

On peut aussi ignorer les signaux pendant leur traitement, mais cela peut créer des pertes de réception.
Enfin, la solution BSD/POSIX où l'on peut bloquer et débloquer la réception de signaux à l'aide du vecteur de masquage (sans pour autant nous assurer de la réception de tous les signaux!!). De plus, en POSIX, le traitement d'un signal comporte une clause de blocage automatique. On indique quels signaux doivent être bloqués pendant le traitement du signal, grâce à un vecteur de masquage dans la structure `sigaction`.

Ceci est le comportement naturel de gestion des interruptions matérielles : on bloque les interruptions de priorité inférieure pendant le traitement d'un interruption.

2. Seconde anomalie des signaux sous System V $<$ V4 : certains appels systèmes peuvent être interrompus et dans ce cas la valeur de retour de l'appel système est -1 (échec). Il faudrait, pour réaliser correctement le modèle d'une interruption logicielle, relancer l'appel système en fin de traitement du signal. (Sous BSD ou POSIX, il est possible de choisir le comportement en cas d'interruption d'un appel système grâce à la fonction `siginterrupt`, c-a-d relancer ou non l'appel système, un appel à `read`, par exemple, peut facilement être interrompu si il nécessite un accès disque).

3. Troisième anomalie des signaux sous ATT : si un signal est ignoré par un processus endormi, celui-ci sera réveillé par le système uniquement pour apprendre qu'il ignore le signal et doit donc être endormi de nouveau. Cette perte de temps est dûe au fait que le vecteur des signaux est dans la zone u et non pas dans le bloc de contrôle du processus.

11.2.1 Le signal SIGCHLD

Le signal SIGCHLD (anciennement SIGCLD) est un signal utilisé pour réveiller un processus dont un des fils vient de mourir. C'est pourquoi il est traité différemment des autres signaux. La réaction à la réception d'un signal SIGCHLD est de repositionner le bit pendant à zéro, et d'ignorer le signal, mais le processus a quand même été réveillé pour cela. L'effet d'un signal SIGCHLD est donc uniquement de réveiller un processus endormi en priorité interruptible.

Si le processus capture les signaux SIGCHLD, il invoque alors la procédure de traitement définie par l'utilisateur comme il le fait pour les autres signaux, ceci en plus du traitement par défaut.

Le traitement normal est lié à la primitive `wait` qui permet de récupérer la valeur de retour (exit status) d'un processus fils. En effet, la primitive `wait` est bloquante et c'est la réception du signal qui va réveiller le processus, et permettre la fin de l'exécution de la primitive `wait`.

Un des problèmes de la gestion de signaux System V est le fait que le signal SIGCHLD est reçu (raised) au moment de la pose d'une fonction de traitement.

Ces propriétés du signal SIGCHLD peuvent induire un bon nombre d'erreurs.

Par exemple, dans le programme suivant nous positionnons une fonction de traitement dans laquelle nous repositionnons la fonction de traitement. Comme sous System V, le comportement par défaut est repositionné pendant le traitement d'un signal. Or le signal est levé à la pose de la fonction de traitement, d'où une explosion de la pile.

```
#include <stdio.h>
#include <unistd.h> /* ancienne norme */
#include <signal.h>

void hand(int sig) {
    signal(sig, hand);
    printf("message qui n'est pas affiche\n");
}

main() {
    if (fork()) { exit(0); /* creation d'un zombi */ }
    signal(SIGCHLD, hand);
    printf("ce printf n'est pas execute\n");
}
```

Sur les HP, un message d'erreur vous informe que la pile est pleine : `stack growth failure`.

Deuxième exemple :

```
#include <signal.h>
#include <sys/wait.h>

int pid, status;

void hand(int sig) {
    printf(" Entree dans le handler \n");
    system("ps -l");                /* affichage avec etat zombi du fils */
    if ((pid = wait(&status)) == -1) /* suppression du fils zombi */
    {
         perror("wait handler ");
         return ;
```

```
    }
    printf(" wait handler  pid: %d    status %d \n", pid, status);
    return;
}

main() {
    signal(SIGCHLD,hand);  /* installation du handler */
    if (fork() ==  0)
    {   /* dans le fils */
        sleep(5);
        exit(2);
    }
/* dans le pere */
    if ((pid = wait(&status)) == -1) /* attente de terminaison du fils */
    {
        perror("wait main ");
        return ;
    }
    printf(" wait main  pid: %d    status %d \n", pid, status);
}
```

résultat :

```
 Entree dans le handler
F S  UID   PID  PPID  C PRI NI     ADDR   SZ  WCHAN    TTY  TIME COMD
1 S  121  6792  6667  0 158 20  81ac180    6  49f5fc  ttys1 0:00 sigchld
1 S  121  6667  6666  0 168 20  81ac700  128 7ffe6000 ttys1 0:00 tcsh
1 Z  121  6793  6792  0 178 20  81bda80    0          ttys1 0:00 sigchld
1 S  121  6794  6792  0 158 20  81ac140   78  4a4774  ttys1 0:00 sh
1 R  121  6795  6794  4 179 20  81bd000   43          ttys1 0:00 ps
wait handler  pid: 6793    status 512     (2 * 256)
wait main: Interrupted system call
```

A la mort du fils, Le père reçoit le signal SIGCHLD (alors qu'il était dans le `wait` du main), puis le handler est executé, et `ps` affiche bien le fils zombi. Ensuite c'est le `wait` du handler qui prend en compte la terminaison du fils. Au retour du handler, l'appel a `wait` du main retourne -1, puisqu'il avait été interrompu par SIGCHLD.

11.3 Manipulation de la pile d'exécution

La primitive

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

sauvegarde un environnement d'exécution, c'est à dire un état de la pile, et si *indicateur* est non nul, sauvegarde le masque de signaux courant. La valeur de retour de cette fonction est zéro quand on fait une sauvegarde, et sinon dépend du paramètre *valeur* de la fonction `siglongjmp`.

```
int siglongjmp(sigjmp_buf env, int valeur);
```

La primitive `siglongjmp` permet de reprendre l'exécution à l'endroit sauvegardé par `sigsetjmp` dans la variable *env*.

Deux remarques : `env` doit avoir été initialisé par `sigsetjmp`, les valeurs de pile placées au-dessus de l'environnement repris sont perdues. L'environnement de pile doit encore exister dans la pile au moment de l'appel, sinon le résultat est indéterminé.

11.4 Quelques exemples d'utilisation

```
/*un exemple de signaux BSD */
#include <stdio.h>
#include <signal.h>
void gots1(int n)   { raise(SIGUSR2);  printf("got  s1(%d) ", n); }
void gots2(int n)   { printf("got  s2(%d) ", n); }

main()
{
    int mask ;
    struct sigvec s1,s2;

    s1.sv_handler = gots1;
    s1.sv_mask = sigmask(SIGUSR1);
    sigvec(SIGUSR1, &s1, NULL);

    s2.sv_handler = gots2;
    s2.sv_mask = sigmask(SIGUSR2);
    sigvec(SIGUSR2, &s2, NULL);

    printf(" sans masquage de SIGUSR2: ")
    raise(SIGUSR1);

    printf(" \n avec masquage de SIGUSR2: " );
    s1.sv_mask = sigmask(SIGUSR2);
    sigvec(SIGUSR1, &s1, NULL);

    raise(SIGUSR1);
}
```

Nous donne les affichages suivant :

```
 sans masquage de SIGUSR2: got  s2(31) got  s1(30)
 avec masquage de SIGUSR2: got  s1(30) got  s2(31)
```

Sous BSD, pas de fonction de manipulation propre des groupes de signaux (on regroupe les signaux par des conjonctions de masques).

Le problème de "l'interruption" des appels système par les signaux est corrigé par la fonction :

```
int  siginterrupt(int sig, int flag);
```

le drapeau `flag` prend comme valeur 0 ou 1, ce qui signifie que les appels systèmes interrompus par un signal seront :
soit relancés avec les mêmes paramètres.
soit retourneront la valeur -1, et dans ce cas la valeur de `errno` est positionnée à `EINTR`.

Certaines fonctions comme `readdir` utilisent des variables statiques, ces fonctions sont dites non réentrantes. Il faut éviter d'appeler ce type de fonctions dans un handler de signal, dans le cas où l'on fait déjà appel à la fonction dans le reste du processus. De la même façon la variable `errno` est unique. Si celle-ci est positionnée dans le `main` mais qu'un signal arrive avant son utilisation, une primitive appelée dans le handler peut en changer la valeur ! (ce problème de réentrance sera vu plus en détail avec les processus multi-activités).

11.4.1 L'appel pause

Fonction de mise en attente de réception d'un signal :

```
pause(void);
```

cette primitive est le standard UNIX d'attente de la réception d'un signal quelconque, BSD propose la primitive suivante :

```
sigpause(int sigmask)
```

qui permet l'attente d'un groupe spécifique de signaux, attention les signaux du masque sont débloqués (c.f. `sigprocmask`).

11.5 La norme POSIX

La norme POSIX ne définit pas le comportement d'interruption des appels systèmes, il faut le spécifier dans la structure de traitement du signal.

Les ensembles de signaux La norme POSIX introduit les ensembles de signaux :
ces ensembles de signaux permettent de dépasser la contrainte classique qui veut que le nombre de signaux soit inférieur ou égal au nombre de bits des entiers de la machine. D'autre part, des fonctions de manipulation de ces ensembles sont fournies et permettent de définir simplement des masques. Ces ensembles de signaux sont du type sigset_t et sont manipulables grâce aux fonctions suivantes :

```
int sigemptyset(sigset_t *ens)           /* raz */
int sigfillset(sigset_t *ens)            /* ens = { 1,2,...,NSIG} */
int sigaddset(sigset_t *ens, int sig)    /* ens = ens + {sig} */
int sigdelset(sigset_t *ens, int sig)     /* ens = ens - {sig } */
```

Ces fonctions retournent -1 en cas d'échec et 0 sinon.

```
int sigismember(sigset_t *ens, int sig);   /* sig appartient à ens ?*/
```

retourne vrai si le signal appartient à l'ensemble.

11.5.1 Le blocage des signaux

La fonction suivante permet de manipuler le masque de signaux du processus :

```
#include <signal.h>
int sigprocmask(int op, const sigset_t  *nouv, sigset_t *anc);
```

L'opération `op` :

SIG_SETMASK affectation du `nouv`eau masque, recupération de la valeur de l'`anc`ien masque.

SIG_BLOCK union des deux ensembles `nouv` et `anc`

SIG_UNBLOCK soustraction `anc - nouv`

On peut savoir si un signal est *pendant* et donc *bloqué* grâce à la fonction :

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

retourne -1 en cas d'échec et 0 sinon et l'ensemble des signaux pendants est stocké à l'adresse `ens`.

11.5.2 sigaction

La structure `sigaction` décrit le comportement utilisé pour le traitement d'un signal :

```
struct sigaction {
        void (*sa_handler) ();
        sigset_t sa_mask;
        int sa_flags;}
```

sa_handler fonction de traitement (ou `SIG_DFL` et `SIG_IGN`)

sa_mask ensemble de signaux supplémentaires à bloquer pendant le traitement

sa_flags différentes options

- **SA_NOCLDSTOP** le signal SIGCHLD n'est pas envoyé à un processus lorsque l'un de ses fils est stoppé.
- **SA_RESETHAND** simulation de l'ancienne méthode de gestion des signaux, pas de blocage du signal pendant le handler et repositionnement du handler par défaut au lancement du handler.
- **SA_RESTART** les appels système interrompus par un signal capté sont relancés au lieu de renvoyer -1. Cet indicateur joue le rôle de l'appel `siginterrupt(sig,0)` des versions BSD.
- **SA_NOCLDWAIT** si le signal est SIGCHLD, ses fils qui se terminent ne deviennent pas zombis. Cet indicateur correspond au comportement des processus pour SIG_IGN dans les versions ATT.

Le positionnement du comportement de reception d'un signal se fait par la primitive `sigaction`. L'installation d'une fonction de traitement du signal SIGCHLD peut avoir pour effet d'envoyer un signal au processus, ceci dans le cas où le processus a des fils zombis, c'est toujours le problème lié à ce signal qui n'a pas le même comportement que les autres signaux.
Un handler positionné par `sigaction` reste jusqu'à ce qu'un autre handler soit positionné, à la différence des versions ATT où le handler par défaut est repositionné automatiquement au début du traitement du signal.

```
#include <signal.h>
int sigaction(int sig,
              const struct sigaction *paction,
              struct sigaction       *paction_precedente);
```

Cette fonction réalise soit une demande d'information. Si le pointeur `paction` est null, on obtient la structure `sigaction` courante. Sinon c'est une demande de modification du comportement.

11.5.3 L'attente d'un signal

En plus de l'appel `pause`, on trouve sous POSIX l'appel `int sigsuspend(const sigset_t *ens) ;` qui permet de réaliser de façons `atomique` les actions suivantes :

- l'installation du masque de blocage défini par `ens` (qui sera repositionné à sa valeur d'origine) à la fin de l'appel,
- mise en attente de la réception d'un signal non bloqué.