Programmation des systèmes

Gestion des signaux

Philippe MARQUET
Philippe.Marquet@lifl.fr
Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille

Licence d'informatique de Lille
mars 2005

Références & remerciements

- *Unix, programmation et communication*
  Jean-Marie Rifflet et Jean-Baptiste Yunès
  Dunod, 2003
- *Introduction aux Systèmes et aux Réseaux*
  Sacha Krakowiak
  Cours de Licence informatique, Université Joseph Fourier, Grenoble
- *The Single Unix Specification*
  The Open Group
  www.unix.org/single_unix_specification/
- `man` section 2

Signaux = interruptions

- Signaux = interruptions logicielles
  - événement asynchrone
  - destiné à un processus
  - émis par un autre processus ou par le système
- Mécanisme de traitant de signaux
  - réaction à la réception d'un signal
  - traitant de signal (*handler*)
  - installation du traitant
  - fonction appelée de manière asynchrone à la réception d'un signal
  - reprise de l'exécution à son point d'interruption
- Communication bas niveau
  - transporte peu d'information : numéro de signal
  - utiliser pour informer d'un événement
  - accord entre émetteur et récepteur sur la sémantique de l'événement

Comportements à la réception d'un signal

- Différents comportements possibles
  - paramétrable pour certains signaux
- Termine l'exécution du processus
  - possibilité de *core dump*
  - image mémoire du processus exploitable par dévermineur
- Suspend l'exécution du processus
  - le processus père est averti (par un autre signal !)
  - peut choisir de faire terminer le fils, de le faire continuer... (en lui envoyant d'autres signaux !)
  - gestion des *jobs* par un shell (`fg`, `bg`...)
- Signal ignoré
  - rien ne se passe...
- Déclenche l'exécution d'une fonction
  - traitant de signal
  - l'exécution normale reprend à la terminaison de la fonction

Principaux signaux

nom	événement	comportement
Terminaison		
`SIGINT`	interruption `<intr>`, `C-c`	terminaison
`SIGQUIT`	interruption `<quit>`, `C-\`	terminaison + core
`SIGHUP`	fin de connexion fin du leader de session	terminaison
`SIGKILL`	terminaison immédiate	terminaison (immuable)
`SIGTERM`	terminaison	terminaison
Fautes		
`SIGFPE`	erreur arithmétique	terminaison + core
`SIGILL`	instruction illégale	terminaison + core
`SIGSEGV`	violation protection mémoire	terminaison + core

Principaux signaux *(cont'd)*

nom	événement	comportement
Divers		
`SIGALARM`	échéance horloge	ignoré
`SIGUSR1` `SIGUSR2`	émis par un processus utilisateur	terminaison
Suspension/reprise		
`SIGTSTP`	suspension `<susp>`, `C-z`	suspension
`SIGSTOP`	signal de suspension	suspension (immuable)
`SIGCHLD`	terminaison ou arrêt d'un fils	ignoré
`SIGCONT`	continuation d'un processus arrêté	reprise (si arrêté !)

Identification d'un signal

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

```
#include <string.h>

char *strsignal(int sig);

extern const char *const sys_signame[NSIG];
```

  - retourne un message associé au signal
  - `#include <signal.h>`

```
void psignal(unsigned sig, const char *msg);
```

  - affiche sur la sortie d'erreur le message `msg` et le message associé au paramètre

Identification d'un signal *(cont'd)*

psignal.c

```
int
main()
{
    int signum;

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

    exit(EXIT_SUCCESS);
}
```

- une exécution a donné

```
% ./psignal
( 0) Signal 0 : Signal 0
( 1)      hup : Hangup
( 2)      int : Interrupt
( 3)     quit : Quit
( 4)      ill : Illegal instruction
( 5)     trap : Trace/BPT trap
( 6)     abrt : Abort trap
( 7)      emt : EMT trap
( 8)      fpe : Floating point exception
( 9)     kill : Killed
(10)      bus : Bus error
(11)     segv : Segmentation fault
(12)      sys : Bad system call
(13)     pipe : Broken pipe
(14)     alrm : Alarm clock
(15)     term : Terminated
(16)      urg : Urgent I/O condition
(17)     stop : Suspended (signal)
(18)     tstp : Suspended
(19)     cont : Continued
(20)     chld : Child exited
(21)     ttin : Stopped (tty input)
(22)     ttou : Stopped (tty output)
(23)       io : I/O possible
(24)     xcpu : Cputime limit exceeded
(25)     xfsz : Filesize limit exceeded
(26)   vtalrm : Virtual timer expired
(27)     prof : Profiling timer expired
(28)    winch : Window size changes
(29)     info : Information request
(30)     usr1 : User defined signal 1
(31)     usr2 : User defined signal 2
```


Envoi de signaux

- Un processus envoie un signal à un autre processus désigné
  - `#include <signal.h>`

    `int kill(pid_t pid, int sig);`
  - nécessite des droits
  - retourne `-1` en cas erreur
  - signal de numéro `0` = pas de signal ; test de validité de `pid`
- Commande `kill`
  - `kill [-signal_name] pid...`
    `kill [-signal_number] pid...`
  - identification par leur nom
  - numéros normalisés
    - `0`, `1` ≡ `SIGHUP`, `9` ≡ `SIGKILL`,
  - utiliser les noms

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 (...à suivre)

Attente d'un signal *(cont'd)*

pause.c

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

    exit(EXIT_SUCCESS);
}
```

```
% ./pause
[15711] pausing...
Killed

% ./pause
[15714] pausing...
Killed

% ./pause
[15719] pausing...
User signal 1

% ./pause
[15722] pausing...
```

```
% kill -0 15711

% kill -KILL 15711

% kill -0 15711
15711: No such process

% kill -9 15714

% kill -USR1 15719

% kill -CHLD 15722
```

États d'un signal

- Un signal est *envoyé*
  - à un processus destinataire
  - par un processus émetteur
- Un signal est *pendant*
  - tant qu'il n'a pas été traité par ce processus destinataire
- Un signal est *délivré*
  - quant 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 traitant de signal d'un signal de même type
  - il ne peut exister qu'un signal pendant d'un type donné
  - des signaux peuvent donc être perdus
- Un signal est *reçu*
  - par ce processus destinataire

Délivrance d'un signal

- Pour chaque type de signal, pour chaque processus, structure :
  - booléen : signal pendant
  - booléen : signal bloqué
  - pointeur vers une fonction traitant de signal
  - masque des signaux temporairement bloqués durant l'exécution du traitant
- Déroulement
  - réception du signal : pendant $\rightsquigarrow$ vrai
  - de manière *asynchrone*
    - si signal $\neg$pendant ou signal bloqué : terminé
    - mise en place du masque (sauvegarde ancien)
    - appel de la fonction traitant de signal
    - reinstallation de l'ancien masque
    - pendant $\rightsquigarrow$ faux
- Conséquence : pas de garantie de non perte de signaux
  - si rafale de signaux

Réglage du comportement à la réception d'un signal

- Différents réglages possibles pour chaque type de signal
  - comportement par défaut
  - ignorance
  - traitant personnalisé
  - masquage
- Comportement par défaut
  - identifié par valeur symbolique `SIG_DFL`
  - propre à chaque type de signal
  - terminaison, ignorance, suspension, etc.
- Ignorance d'un signal
  - identifié par valeur symbolique `SIG_IGN`
  - le signal est bien délivré, mais le comportement est de ne rien faire

Réglage du comportement à la réception d'un signal *(cont'd)*

- Traitant personnalisé
  - exécuté par le processus destinataire du signal
  - fonction du code du programme
  - de prototype

    ```
    void handler(int signum);
    ```

    donc de type

    ```
    void (*phandler)(int);
    ```

    paramètre : simple identification du type du signal
  - retourne au code interrompu du programme après exécution
  - signal pour cause de fautes
    - erreur arithmétique, instruction illégale, ou violation protection mémoire
    - le traitant de signal doit avoir réglé le problème...

Réglage du comportement à la réception d'un signal (cont'd)

- Masquage de signaux
  - pour chacun des signaux
  - indiquer si on le bloque ou non
  - trois opérations
    - manipulation d'ensemble de signaux
    - installation manuelle d'un masque de blocage des signaux
    - identification du masque temporaire des signaux bloqués lors de l'exécution du traitant d'un signal

Manipulation d'ensembles de signaux

- Un 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
  - identifie les signaux qui seront bloqués
  - ```
    #include <signal.h>

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

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
  - ```
    #include <signal.h>

    int sigpending(sigset_t *pending);
    ```


Installation d'un masque de blocage (cont'd)

sigpdg.c

```
int
main ()
{
    sigset_t sset;

    pr_prending("initially");

    sigemptyset(&sset);
    sigaddset(&sset, SIGUSR1);
    sigaddset(&sset, SIGINT);

    sigprocmask(SIG_SETMASK,&sset, 0);

    kill(getpid(), SIGUSR1);
    kill(getpid(), SIGUSR1);

    pr_prending("after kill");

    sigemptyset(&sset);
    sigprocmask(SIG_SETMASK,&sset, 0);

    fprintf(stderr, "bye\n");
    exit(EXIT_SUCCESS);
}
```

```
static void
pr_prending(const char *str)
{
    sigset_t sset;
    int sig;

    sigpending(&sset);

    fprintf(stderr, "%s, pendings:", str);
    for(sig=1; sig<NSIG; sig++)
        if (sigismember(&sset, sig))
            fprintf(stderr, " %d", sig);
    fputc('\n', stderr);
}
```

```
% ./sigpdg
initially, pendings:
after kill, pendings: 30
User signal 1
```

Traitant de signal personnalisé

- Structure `struct sigaction`
  - ```
    struct sigaction {
        void (*sa_handler)();
        sigset_t sa_mask;
        int sa_flags;
    };
    ```
  - la fonction `sa_handler` (ou `SIG_DFL`, ou `SIG_IGN`) est le traitant
  - les signaux `sa_mask` seront masqués durant l'exécution de la fonction
  - (`sa_falgs` identifie quelques options)
- Primitive `sigaction()`
  - ```
    #include <signal.h>

    int sigaction (int sig, const struct sigaction *new,
                   struct sigaction *old);
    ```
  - installe le traitant `new`
  - récupère l'ancien traitant dans `old`

Traitant de signal personnalisé *(cont'd)*

nocc.c

```
#define NSIGMAX 5

static void
set_default()
{
    struct sigaction sa;

    sa.sa_handler = SIG_DFL;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;

    sigaction(SIGINT, &sa,
              (struct sigaction *) 0);
}

static 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,
              (struct sigaction *) 0);

    for(;;)
        pause();

    fprintf(stderr, "bye\n");
    exit(EXIT_SUCCESS);
}
```

```
% ./nocc
^C C-c won't kill me
^C C-c won't kill me
^C C-c won't kill me
^C C-c won't kill me
^C C-c won't kill me
^C unless you insist...
^C
%
```


Traitant de signal personnalisé *(cont'd)*

- Possible perte de signaux

rafale.c

```
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,
              (struct sigaction *) 0);
    sigaction(SIGUSR1, &sa,
              (struct sigaction *) 0);

    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);
}
```

```
static 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:
            ;
    }
}
```

```
% ./rafale 5000
bye
^Csignaux recus: 5

% ./rafale 50000
bye
^Csignaux recus: 12437

% ./rafale 50000
^Csignaux recus: 14981
```


Traitant de signal personnalisé *(cont'd)*

- Correction de faute nécessaire

nocd.c

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


int
main ()
{
    struct sigaction sa;

    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;

    sigaction(SIGSEGV, &sa,
              (struct sigaction *) 0);

    {
        char *c = ((char *) sbrk(0))
                  + getpagesize() + 1;
        *c = 'a';

    }

    fprintf(stderr, "bye\n");
    exit(EXIT_SUCCESS);
}
```

```
% ./nocd
I will not core dumped...
I will not core dumped...
I will not core dumped...
I will not core dumped...
I will not core dumped...
I will not core dumped...
I will not core dumped...
I will not core dumped...
I will not core dumped...
I will not core dumped...
I will not core dumped...
```

Ancienne interface `signal()`

- Alternative originelle à `sigaction()`
- À ne pas utiliser !
  - même si a priori plus simple
  - on peut rencontrer des codes l'utilisant
- Joli prototype
  - `#include <signal.h>`

    `void (*signal(int sig, void (*func)(int)))(int);`

    (retourne l'ancienne fonction)
- À la délivrance d'un signal
  - le comportement par défaut (peut être) est réinstallé
- Palliatif : nouvelle installation manuelle
  - ```
    void handler(int sig) {
        signal(sig, handler);
        ...
    }
    ```
  - si un signal est délivré entre l'appel de `handler()` et `signal()`...

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>

    int alarm(unsigned seconds);
    ```
  - un seul minuteur par processus
  - nouvel armement annule le précédent
  - délai nul supprime la requête

Temporisation *(cont'd)*

quizz.c

```
#define LINE_MAX 128
#define DELAY 10

static 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, (struct sigaction *) 0);

    printf("Reponse ? ");
    alarm(DELAY);
    if (fgets(answer, LINE_MAX, stdin)) {
        alarm(0);
        printf("ok...\n");
    }

    exit(EXIT_SUCCESS);
}
```

```
% ./quizz
Reponse ? OUI
ok...
% ./quizz
Reponse ?
trop tard..
%
```


Temporisations avancées

- Temporisation par `alarm()`
  - temps-réel *wall-clock time*
  - résolution, à la seconde
- Structure `struct timeval` (`sys/time.h`)
  - ```
    struct timeval {
        time_t    tv_sec;       /* seconds */
        long int  tv_usec;      /* microseconds */
    };
    ```
- Structure `struct itemerval` (`sys/time.h`)
  - ```
    struct itemerval {
        struct timeval it_interval;
        struct timeval it_value;
    };
    ```
  - timer périodique
  - spécifie une échéance à `it_value`
  - puis une toutes les `it_interval`
  - `it_value` à 0 : annulation
  - `it_interval` à 0 : pas de réarmement

Temporisations avancées *(cont'd)*

- Trois temporisations

itimer	temporisation	signalisation
`ITIMER_REAL`	temps réel	`SIGALRM`
`ITIMER_VIRTUAL`	temps processeur mode utilisateur	`SIGVTALRM`
`ITIMER_PROF`	temps processeur total	`SIGPROF`

- Armement du minuteur
  - `#include <sys/time.h>`

```
int setitimer(int itimer, const struct itimerval *new,
              struct itimerval *old);
```

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

Temporisations avancées *(cont'd)*

itmr.c

```
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,
              (struct sigaction *) 0);

    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,
              (struct itimerval *)0);

    for (;;)
        i++;

    exit(EXIT_SUCCESS);
}
```

```
static struct tms start, end;

static float
tics_to_seconds(clock_t tics)
{
    return tics/(float)sysconf(_SC_CLK_TCK);
}

static void
handler(int sig)
{
    times(&end);
    printf("%.6f\n",
           tics_to_seconds(end.tms_utime
                           - start.tms_utime));
    times(&start);
}
```

```
 ./itmr
0.200000
0.500000
0.500000
0.500000
0.500000
0.500000
0.500000
0.500000
```


Terminaison / blocage des fils

- 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
  - passage du fils dans l'état bloqué par réception de `SIGSTOP` ou `SIGTSTP`
  - signal `SIGCHLD`
  - comportement par défaut : ignorance
    - traitement...
    - le relancer par un signal `SIGCONT`

Terminaison / blocage des fils *(cont'd)*

cont.c

```
static pid_t fils;

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 void
handler(int sig)
{
    printf("[%d] ai recu signal %d\n",
           getpid(), sig);
    kill(fils, SIGCONT);
}
```

```
% ./cont
[911] kill(911, 17)
[910] ai recu signal 20
[910] ai recu signal 20
[910] ai recu signal 20
```

Terminaison / blocage des fils *(cont'd)*

stop0.c

```
static pid_t fils;

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 void
handler(int sig)
{
    printf("[%d] ai recu signal %d\n",
           getpid(), sig);
    system("ps");
}
```

```
% ./stop0
[1292] ai recu signal 20
  PID  TT  STAT      TIME COMMAND
  706  p1  Ss+    0:00.10 -tcsh
  [...]
[1292] ai recu signal 20
  PID  TT  STAT      TIME COMMAND
  706  p1  Ss+    0:00.10 -tcsh
  [...]
[1292] ai recu signal 20
  PID  TT  STAT      TIME COMMAND
  706  p1  Ss+    0:00.10 -tcsh
  [...]
[1292] ai recu signal 20
  PID  TT  STAT      TIME COMMAND
  706  p1  Ss+    0:00.10 -tcsh
  [...]
[1292] ai recu signal 20
  PID  TT  STAT      TIME COMMAND
```


Terminaison / blocage des fils *(cont'd)*

stop.c

```
static struct sigaction sa;

static void
handler(int sig)
{
    printf("[%d] ai recu signal %d\n",
           getpid(), sig);

    sa.sa_handler = SIG_DFL;
    sigaction(SIGCHLD, &sa, NULL);

    system("ps");

    sa.sa_handler = handler;
    sigaction(SIGCHLD, &sa, NULL);

    kill(fils, SIGCONT);
}
```

```
% ./stop
[2201] kill(2201, 17)
[2200] ai recu signal 20
  PID  TT  STAT      TIME COMMAND
  706  p1  Ss+    0:00.10 -tcsh
  793  p1  S      0:41.49 emacs
 2200 std  S+     0:00.00 ./stop
 2201 std  S+     0:00.01 ./stop
[2200] ai recu signal 20
  PID  TT  STAT      TIME COMMAND
  706  p1  Ss+    0:00.10 -tcsh
  793  p1  S      0:41.49 emacs
 2200 std  S+     0:00.00 ./stop
 2201 std  S+     0:00.01 ./stop
[2200] ai recu signal 20
  PID  TT  STAT      TIME COMMAND
  706  p1  Ss+    0:00.10 -tcsh
  793  p1  S      0:41.49 emacs
 2200 std  S+     0:00.00 ./stop
 2201 std  Z+     0:00.00 (stop)
```


Terminaison / blocage des fils *(cont'd)*

hdlwait.c ***(début)***

```
static void
f(int seconds, const char *fname)
{
    printf("fonction %s() executee par le processus %d\n",
           fname, getpid()) ;
    sleep(seconds) ;
    printf("fonction %s() terminee\n",
           fname) ;
}

static void fa(void) { f(12, "fa"); }
static void fb(void) { f(2, "fb"); }
static void fc(void) { f(3, "fc"); }

static void
trif(void(*f1)(void), void(*f2)(void), void(*f3)(void))
{
    /* lance 3 processus fils pour
       executer les 3 fonctions */
    ...
}
```

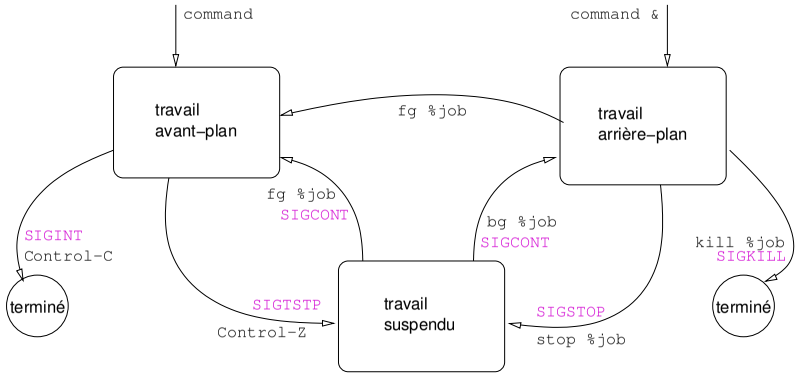

Terminaison / blocage des fils *(cont'd)*

hdlwait.c ***(suite et fin)***

```
static void
hdlwait(int sig)
{
    pid_t pidz;
    pidz = wait(NULL);
    printf("[%d] ai traite terminaison du fils %d\n", getpid(), pidz);
}

int
main()
{
    struct sigaction sa;

    sa.sa_handler = hdlwait;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;

    sigaction(SIGCHLD, &sa, NULL);

    trif(fa, fb, fc);

    for(;;) {
        printf("main at work...\n");
        sleep(2);
    }

    exit(EXIT_SUCCESS);
}
```

```
% ./hdlwait
fonction fa() executee par le processus 2557
fonction fc() executee par le processus 2559
main at work...
fonction fb() executee par le processus 2558
main at work...
fonction fb() terminee
[2556] ai traite terminaison du fils 2558
main at work...
fonction fc() terminee
[2556] ai traite terminaison du fils 2559
main at work...
main at work...
main at work...
main at work...
main at work...
fonction fa() terminee
[2556] ai traite terminaison du fils 2557
main at work...
main at work...
main at work...
```

Gestion des travaux



- Travaux ou jobs = processus
  - arrière-plan, *background*
  - avant-plan, *foreground*
  - suspendu, *suspended*
- Commandes gestion de travaux
  - gérées par des signaux

Session de processus

- Partitionnement des processus du système en sessions
  - un processus appartient donc à exactement une session
  - par défaut, la session de son père
  - par exemple : la session de l'ensemble des processus lancés depuis un shell
- Processus *leader* d'une session
  - processus qui crée une session

```
#include <unistd.h>

pid_t setsid(void);
```

  - session identifiée par le PID du processus leader
- Session et terminaux
  - le leader acquiert un terminal de contrôle en ouvrant `/dev/tty`
  - les processus de la session pourront être informés par des signaux de la frappe de `<quit>`, `<intr>`, etc
  - à la terminaison du leader, les processus de la session reçoivent `SIGHUP`

Groupes de processus

- Partitionnement d'une session en groupes de processus
  - un processus appartient donc à exactement un groupe
  - distinguer les processus interactifs des processus en arrière-plan
- Groupe des processus en avant-plan
  - un unique groupe des processus en avant-plan par session
  - processus peuvent accéder au terminal
  - processus destinataires des signaux `SIGINT`, `SIGQUIT`, et `SIGTSTP` issus des caractères `<intr>`, `<quit>`, `<susp>`
- Des groupes de processus en arrière-plan
  - processus en arrière plan ne peuvent accéder au terminal (sinon reçoivent `SIGSTOP`)
  - processus en arrière-plan ne reçoivent pas les signaux `SIGINT`, `SIGQUIT`, et `SIGTSTP` issus des caractères `<intr>`, `<quit>`, `<susp>`

Groupes de processus *(cont'd)*

- Processus *leader* du groupe
  - processus qui crée le groupe

```
#include <unistd.h>

pid_t getpgrp(void);
```

  - groupe identifié par le PID du processus leader
- Rattacher un processus à un groupe
  - `#include <unistd.h>`

```
int setpgid(pid_t pid, pid_t gpid);
```

  - rattache le processus `pid` au groupe `gpid`

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...
- Processus 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`
  - au programme de le relancer

Contrôle du point de reprise *(cont'd)*

intrwait.c

```
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);
}
```

```
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));
}
```

```
% ./intrwait
handler wait: pidz 14916, status 2
main wait: Interrupted system call
```


Contrôle du point de reprise *(cont'd)*

- 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, ...);
```


Contrôle du point de reprise *(cont'd)*

intrwaitr.c

```
int
main()
{
    struct sigaction sa;
    pid_t pidz;
    int status;

    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;

    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);
}
```

```
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));
}
```

```
% ./intrwaitr
handler wait: pidz 14943, status 2
main wait: No child processes
```