Programmation Multitaches - TP1 & 2

Rémi Maubanc

08 Octobre 2021

1 TP1 - Les signaux

1.1 Ignorer les signaux

1. On compte 31 signaux utiles recensés selon l'affichage donné par *strsignal* . Les signaux marqués comme *Unknow Signal X* sont des signaux reservés pour un ajout spécifique (par un développeur pour son programme par exemple). Voir *Figure 1* (Fichier associé : *TP1/1_print_signal.c*).

2. Les signaux SIGKILL et SIGSTOP ne peuvent pas être ignorés. En effet, si un processus arrive à ignorer tous les signaux, il n'est plus contrôlable par le noyau. Si ce dernier est malveillant, il peut endommager le système tout en étant inarrêtable. (Fichier associé *TP1/2_catch_signal.c*)

3. Voir Listing 1

4. On peut envoyer les signaux via le programme *htop* et confirmer la gestion de ces signaux par notre programme. La commande *kill* envoi le signal *SIGTERM*. Ce n'est pas le même signal que la commande *Ctrl+C* qui envoi le signal *SIGINT*.

```
hyperion@LAPTOP-0V3CQIBK:~/PM_grit/TP1$ gcc 1_print_signal.c
hyperion@LAPTOP-0V3CQIBK:~/PM_grit/TP1$ ./a.out
0: Unknown signal 0
1: Hangup
2: Interrupt
3: Quit
4: Illegal instruction
5: Trace/breakpoint trap
6: Aborted
7: Bus error
8: Floating point exception
9: Killed
10: User defined signal 1
11: Segmentation fault
12: User defined signal 2
13: Broken pipe
14: Alarm clock
15: Terminated
16: Stack fault
17: Child exited
18: Continued
19: Stopped (signal)
20: Stopped
21: Stopped (tty input)
22: Stopped (tty output)
23: Urgent I/O condition
24: CPU time limit exceeded
25: File size limit exceeded
26: Virtual timer expired
27: Profiling timer expired
28: Window changed
29: I/O possible
30: Power failure
31: Bad system call
32: Unknown signal 32
33: Unknown signal 33
34: Real-time signal 0
35: Real-time signal 1
36: Real-time signal 2
```

Figure 1: Affichage de la liste des signaux

```c
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>


static int loop = 1;

void sig_handler(int sign)
{
    printf("received %s\n", strsignal(sign));
}

int main(void)
{
    if (signal(SIGHUP, sig_handler) == SIG_ERR)
        return 1;
```

```
18      if (signal(SIGINT, sig_handler) == SIG_ERR)
19          return 2;
20      [...]
21      while (loop) // Ajout d'une boucle infinie pour tester les signaux
22          sleep(1);
23
24      return 0;
25  }
```

Listing 1: Ajout d'une boucle pour tester les signaux

1.2 Traitement spécifique des signaux

1. Pour réussir à envoyer pléthore de signaux à mon programme pour le tester, je démarre le logiciel *htop* dans un autre terminal et j'identifie mon programme dans la liste. Il ne me reste qu'à appuyer sur F9 et de sélectionner le signal à lui envoyer. Voir *Figure 2*.

2. A l'instar de la première question, on peut soit utiliser *kill* $< pid >$ soit sélectionner le processus dans *htop* et lui envoyer le signal *SIGKILL*.
   (Fichier associé : *TP1/3_catch_print_signal.c*)

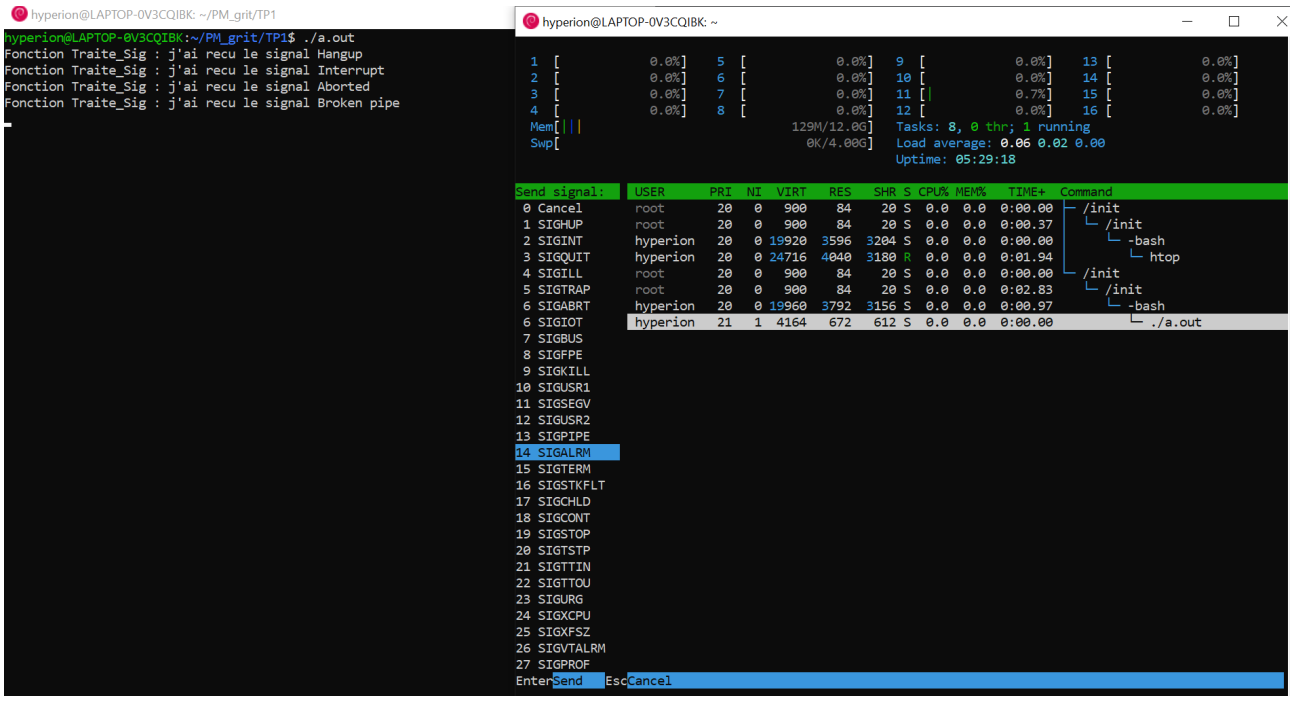

Figure 2: Test des signaux avec htop

1.3 Utilisation de SIGUSR1 et SIGUSR2

Dans le programme, un traitement spécifique est appliqué pour chaque signal intercepté. Il suffit de modifier les conditions pour les deux signaux *SIGUSR1* et *SIGUSR2* en remplaçant la fonction appelée. (voir Listing 2). On peut stopper le programme uniquement via un autre terminal et non avec *Ctrl+Z*. Une fois mis en pause, il suffit de revenir dans le terminal et de taper la commande *fg* pour le réveiller et le remettre au premier plan. (Fichier associé : *TP1/4_catch_sigusr_signal.c*)

```c
#include <stdio.h>
#include <unistd.h>
static int loop = 1;

void fonc(int sign) // Affiche le numero du signal
{
    printf("Numero du signal: %d\n", sign);
}
void fonc1(int sign)
{
    fonc(sign);
    system("who");
}
void fonc2(int sign)
{
    fonc(sign);
    system ("df .");
}

int main(void)
{
    printf("%d\n", getpid()); // Affiche le PID
    if (signal(SIGHUP, fonc) == SIG_ERR)
        return 1;
    [...]
    if (signal(SIGFPE, fonc) == SIG_ERR)
        return 9;
    // Le 10e signal n'est pas interceptable (SIGKILL)
    if (signal(SIGUSR1, fonc1) == SIG_ERR)
        return 11;
    [...]
```

```
    if (signal(SIGUSR2, fonc2) == SIG_ERR)
        return 13;
    [...]
    while (loop)
        sleep(1);
    return 0;
}
```

Listing 2: Gestion de SIGUSR1 et SIGUSR2

1.4 Signaux et Sleep

Lorsque l'on envoi un signal, l'attente du sleep est ignoré pour un tour et affiche la chaine de caractère *Je dors* directement après avoir executé la fonction de gestion du signal. (Fichier associé : *TP1/5_catch_sleep_signal.c*)

1.5 Traitement de SIGFPE

1. Lorsque que le signal est intercepté, la fonction *Traite_FPE* affiche le message d'erreur, puis le programme reprend juste avant l'erreur et ignore totalement le sleep. Ainsi, on affiche en continue le message de *Traite_FPE* sans tenir compte du sleep. (Fichier associé : *TP1/6_catch_sigfpe_signal.c*)

2. Voir Listing 3
   (Fichier associé : *TP1/7_catch_sigjmp_signal.c*)

```
#include [...]
static int loop = 1;

void fonc(int sign) // Affiche le numero du signal
{
    printf("Numero du signal: %d\n", sign);
}
void traiteFPE(int sign)
{
    printf("Detection d'une erreur\n");
    siglongjmp(mark, -1);
}

```

```c
int main(void) {
    printf("%d\n", getpid()); // Affiche le PID
    if (signal(SIGHUP, fonc) == SIG_ERR)
        return 1;
    [...]
    if (signal(SIGFPE, traiteFPE) == SIG_ERR)
        return 9;

    while (loop) {
        if (sigsetjmp(mark, 1) != 0)
            printf("Reprise sur erreur\n");
        sleep(1);
        int c = 1 / 0;
    }
    return 0;
}
```

Listing 3: Gestion de SIGFPE

2 TP2 - Les tubes

2.1 Exercice préliminaire

Voir le code *TP2/1_exo_pipe.c* et les commentaires.

2.2 Le Trone de Fer

Voir le code *TP2/2_iron_trone.c* et les commentaires.