Les signaux et les signaux temps-réel

52010 Schellekens Arnaud

16 Nomvembre 2020

Table des matières

1 Introduction

La gestion des signaux entre processus peut être très utile dans beaucoup de cas concrets. Le principe est simple : un processus peut envoyer des signaux à un autre processus ou lui-même et le destinataire doit immédiatement réagir. Il peut réagir de différentes façons en ignorant simplement le signal ou en exécutant le traitement associé à celui-ci. Nous avions déjà vu que le noyau en utilisait pour arrêter notre programme ou encore lorque l'on abordait le fork où le processus fils notifiait le père de sa mort. [1].

1.1 Généralités

Il existe 32 signaux différents ayant un numéro associé et un nom défini sauf le signal numéro 0 qui est un peu plus spécial. Ceux-ci se trouvent dans le fichier d'en-tête <signal.h>. Pour utiliser ces signaux, il faut toujours utiliser le nom symbolique du signal et non son numéro car le numéro d'un même signal peut varier en fonction du système ou de la machine.Il existe aussi les signaux temps-réel qui ne sont pas supportés sur tous les systèmes. Afin de savoir si la machine supporte les signaux temps-réel, il faut utiliser la constante symbolique _POSIX_REALTIME_SIGNALS qui est définie dans $< unistd.h >$. Ensuite, nous ne savons pas combien de signaux sont disponibles et nous devons éviter d'utiliser leur numéro pour les utiliser. Pour pouvoir y parvenir, nous avons à notre disposition deux constantes symboliques importantes : SIGRTMIN et SIGRTMAX. Ils représentent respectivement le numéro du plus petit signal et du plus grand signal temps-réel. Grâce à ces deux valeurs, nous pouvons accéder à tous les signaux temps-réel en utilisant les deux bornes. Voici un exemple introduisant l'utilisation des signaux temps-réel :

Code

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

#if defined(_POSIX_REALTIME_SIGNALS)
#error "Pas de signaux temps-réel disponibles"
#endif

#define SIGRT0 (SIGRTMIN)
#define SIGRT1 (SIGRTMIN + 1)
#define SIGRT13 (SIGRTMIN + 13)
#define NB_SIGRT_NEEDED 14

int main()
{
    if ((SIGRTMAX - SIGRTMIN + 1) < NB_SIGRT_NEEDED)
        printf("Pas assez de signaux temps-réel \n");
    printf("Les signaux temps réels commencent à partir de %d \n", SIGRTMIN);
    printf("Les signaux temps réels finissent à %d \n", SIGRTMAX);
    printf("Nombre de signaux disponibles %d \n", SIGRTMAX - SIGRTMIN);
    printf("Nombre de signaux nécessaires %d \n", NB_SIGRT_NEEDED);
}
```

1. Nous ne rapellerons pas tout ce qui a déjà été vu au cours.

Output

```
Les signaux temps réels commencent à partir de 34
Les signaux temps réels finissent à 64
Nombre de signaux disponibles 30
Nombre de signaux nécessaires 14
```

Une bonne pratique lorsque nous utilisons des signaux temps-réel est d'utiliser des macros afin de nommer le signal et d'améliorer la lisibilité du code. Une autre pratique est de vérifier si la machine possède suffisamment de signaux pour notre programme grâce à ses macros que nous avons définies.

Nous remarquons aussi que les signaux temps-réel commencent à la valeur 34 et non 32 comme on pourrait s'y attendre. Nous pouvons aussi observer que 30 signaux sont à notre disposition au lieu des 32 disponibles sur la machine. Lorsque nous affichons la liste complète des signaux disponibles sur la machine, nous voyons le même problème. Où sont passés les signaux 0, 32 et 33 ?

```
 1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP
 6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN
    +13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52)
    SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57)
    SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX
    -2
63) SIGRTMAX-1 64) SIGRTMAX
```

En lisant la documentation sur les signaux temps-réel, nous pouvons lire que l'implémentation des threads glibc POSIX utilise en interne deux ou trois de ces signaux. Pour éviter des conflits entre nos processus et ceux en interne, la valeur des macros SIGRTMIN est réajustée. C'est un argument supplémentaire de ne jamais utiliser les signaux par leur valeur mais en utilisant la constante symbolique SIGRTMIN.

En ce qui concerne le signal 0, lorsque nous écrivons kill(pid, 0), la documentation nous dit qu'aucun signal n'est envoyé mais que les conditions d'erreurs sont vérifées. Cela signifie que le processus visé ne recevra aucun signal mais que le destinateur peut recevoir une erreur. Cela a son utilité pour vérifier si un programme existe ou non.

2 Cas concrets

Une fois les bases vues, passons à la pratique avec quelques exemples de cas concrets utilisant les signaux et ses propriétés. Nous verrons que le champ des possibilités est vaste et qu'une solution ne

sera pas toujours suffisante pour couvrir toutes les possibilités.

2.1 Un read non bloquant

L'idée est simple, lire au clavier pendant une certaine durée. L'appel système read étant bloquant tant que rien n'est lu au clavier, nous aimerions que notre programme puisse reprendre après une certaine durée. Cela peut être utile sur un serveur par exemple. Le serveur attend une réponse du client et s'il n'a pas répondu endéans les 5 secondes, une valeur par défaut sera utilisée.

Avant d'écrire le code, nous devons voir comment réagissent les appels systèmes lents tel que read lorsque le programme reçoit un signal. En effet, lorsqu'un signal est reçu et que l'ordonnanceur donne la main à notre processus, l'appel système est intérrompu et le gestionnaire de signal est appelé. Une fois le signal traité, les appels systèmes lents nous disent que ceux-ci redémarrent si rien ne s'est passé. Hors, ce n'est pas ce que nous recherchons, nous ne voulons pas relire au clavier après que notre signal ait été envoyé et traité. Pour cela, il est possible de modifier ce comportement grâce à la fonction *siginterrupt* qui nous permet de préciser si l'appel système intérrompu doit ou non reprendre.

Ecrivons maintenant un programme simple qui lit un caractère au clavier pendant 5 secondes. Si le caractère ENTER a été lu, un message disant "Gagné" sera envoyé. Si rien n'a été lu avant ces 5 secondes ou que ce n'est pas bon le bon caractère, un message disant "Perdu" sera envoyé. Ici nous utiliserons la fonction alarm(5) qui envoie un signal SIGALRM à notre programme.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4  #include <unistd.h>
5  #include <stdbool.h>
6
7  static void sig_handler(int);
8
9  ssize_t myreadtimer(int fd, void *buf, size_t count, unsigned timer);
10
11 int main(int argc, char *argv[])
12 {
13   char c;
14   printf("Tapez return en moins de 5 secondes ! \n");
15   //on initialise le gestionnaire de signal pour traiter le signal sigalarm
16   if (signal(SIGALRM, sig_handler) == SIG_ERR)
17   {
18     perror("signal");
19     exit(EXIT_FAILURE);
20   }
21   //on modifie le comportement d'un appel système interrompu
22   //par un signal sigalarm pour que l'appel système renvoit -1 s'il n'y a eu
23   //aucun transfert de données
24   if (siginterrupt(SIGALRM, true) < 0)
25   {
26     perror("siginterrupt");
27     exit(EXIT_FAILURE);
28   }
```

```
29    //on lance la minuterie
30    alarm(5);
31    //on lit au clavier
32    int r = read(STDIN_FILENO, &c, 1);
33    //le programme est bloqué par read
34    //il sera débloqué soit par une lecture au clavier, soit par le signal sigalarm de la minuterie
35    if ((r == 1) && (c == '\n'))
36    {
37      printf("Gagné ! \n");
38      exit(EXIT_SUCCESS);
39    }
40    printf("Perdu ! \n");
41    exit(EXIT_FAILURE);
42  }
43
44  static void sig_handler(int signum)
45  {
46    printf("Signal sigalarm traité ! \n");
47  }
48
49  //On peut alors par exemple écrire une fonction comme read avec un paramètre
50  //supplémentaire qui serait la durée de la lecture au clavier
51  ssize_t myreadtimer(int fd, void *buf, size_t count, unsigned timer)
52  {
53    if (siginterrupt(SIGALRM, true) < 0)
54    {
55      perror("siginterrupt");
56      exit(EXIT_FAILURE);
57    }
58    alarm(timer);
59    ssize_t r = read(fd, buf, count);
60    alarm(0);
61    return r;
62  }
```

Nous pourrions penser qu'aux premiers abords, ce programme n'ait pas de bugs. Néanmoins il existe un cas où l'ordonnanceur pourrait créer quelques problèmes. Imaginons le cas où notre programme arrive à la ligne 30 et exécute l'instruction alarm(5) qui a pour but de lancer la minuterie. Ensuite l'ordonnanceur donne la main à un autre programme et, pour une raison ou l'autre, le système est fort chargé. Comme le système est chargé, notre programme ne reprend qu'après la sixième seconde. Lorsqu'il a à nouveau la main, le signal SIGALRM a déjà été envoyé car 5 secondes s'étaient écoulées donc le traitement doit d'abord être éxécuté. Une fois, le traitement du signal fait, notre code peut continuer et éxécuter l'instruction suivante read(). Sauf que la minuterie étant déjà passée, notre processus sera bloqué par l'appel système et il ne recevra jamais le signal SIGALRM.

Pour pallier à ce problème, nous devons rajouter quelques instructions supplémentaires afin d'éviter de faire un read si la minuterie est déjà passée. Pour cela, nous allons utiliser les fonctions sigsetjmp et siglongjmp contenue dans le fichier d'en-tête $< setjmp.h >$.

Un premier appel à sigsetjmp enregistre l'environnement dans une variable et retourne 0. Ensuite, un appel à siglongjmp en donnant l'environnement donné par sigsetjmp nous remet à l'instruction sigsetjmp et l'éxécute à nouveau mais retourne cette fois-ci une valeur différente de 0. Cela nous permet donc de revenir en arrière mais de ne pas forcément refaire les mêmes instructions car nous pouvons utiliser une condition afin d'éviter cela.

Modifions un peu le code pour régler le problème.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4  #include <unistd.h>
5  #include <stdbool.h>
6  #include <setjmp.h>
7
8  sigjmp_buf env;
9
10 static void sig_handler(int);
11
12 int main(int argc, char *argv[])
13 {
14     char c;
15     printf("Tapez return en moins de 5 secondes ! \n");
16     if (signal(SIGALRM, sig_handler) == SIG_ERR)
17     {
18         perror("signal");
19         exit(EXIT_FAILURE);
20     }
21     if (siginterrupt(SIGALRM, true) < 0)
22     {
23         perror("siginterrupt");
24         exit(EXIT_FAILURE);
25     }
26     int r = 0;
27     //on enregistre l'environnement
28     if (sigsetjmp(env, 1) == 0)
29     {
30         //on passe une première fois dans le if car le premier appel à setjmp renvoie toujours 0
31         //le second appel à setjmp par longjmp renvoie lui une valeur non nulle
32         //sig_handler n'a pas encore été appelé
33         alarm(5);
34         //même si le programme était bloqué après l'instanciation de la minuterie et avant la
               lecture du clavier.
35         //on n'est pas bloqué car le signal sigalarm doit être traité avant que le code continue
               donc on finit dans le sig_handler
36         //qui nous emmène dans le else et on n'est pas bloqué car le read n'est jamais lancé.
37         r = read(STDIN_FILENO, &c, 1);
38     }
39     else
40     {
41         // sig_handler a déjà été exécuté
```

```
42              // le délai a déjà expiré, inutile de faire read
43      }
44      if ((r == 1) && (c == '\n'))
45      {
46          printf("Gagné ! \n");
47          exit(EXIT_SUCCESS);
48      }
49      else
50      {
51          printf("Perdu ! \n");
52          exit(EXIT_FAILURE);
53      }
54  }
55
56  static void sig_handler(int signum)
57  {
58      siglongjmp(env, 1);
59  }
```

Le comportement classique reste le même, notre read sera bien intérrompu par un signal ou par un caractère lu. Néanmoins, si l'alarme se déclenche après notre read, notre programme ne lancera pas le read grâce à sigsetjmp.

Nous avons vu que les signaux pouvaient vite complexifier de simples codes. Voyons un autre cas qui peut poser quelques problèmes lorsque nous ne faisons pas attention.

2.2 Gérer une zone critique

Lorsque nous utilisons un gestionnaire de signal, nous devons faire attention aux variables que nous manipulons. En effet, le traitement d'un signal utilise le même contexte que notre main(). C'est pour cela qu'il est déconseillé d'utiliser des fonctions utilisant des variables statiques ou globales dans un gestionnaire car il pourrait y avoir des conflits. Par exemple, si notre processus fait appel à malloc qui utilise des variables statiques, que notre code est intérrompu lors de l'appel à malloc, et que notre traitement du signal fait lui aussi un malloc. Les variables statiques seront modifiées par le malloc du traitement du signal et le second malloc du code qui reprendra où il s'est arrêté aura des données modifiées qui seront incohérentes. Un autre exemple que nous allons illustrer est la gestion d'une zone critique avec une variable globale.

Voici un simple exemple qui utilise une variable globale comme diviseur d'un nombre. La gestion du signal a pour but de modifier ce diviseur et de le mettre à 0.

Code

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5
6  int diviseur;
7
```

```c
void traitement(int signal)
{
    diviseur = 0;
    printf("Diviseur vaut désormais 0 \n");
}

int main()
{
    struct sigaction action;
    action.sa_handler = traitement;
    sigaction(SIGUSR1, &action, NULL);
    long int val = 10000000000;
    diviseur = 10;
    if (fork() == 0)
    {
        sleep(4);
        printf("Signal SIGUSR1 envoyé \n");
        kill(getppid(), SIGUSR1);
        exit(0);
    }
    else
    {
        while (val != 100)
        {
            val = val / diviseur;
            printf("%ld \n", val);
            sleep(1);
        }
    }
}
```

Output

```
1000000000
100000000
10000000
1000000
Signal SIGUSR1 envoyé
Diviseur vaut désormais 0
Floating point exception (core dumped)
```

Lorsque nous lançons ce programme, nous pouvons voir que le traitement du signal modifie la valeur à 0 et provoque une division par zéro. Une façon de résoudre ce problème est de protéger cette zone critique du programme et de bloquer l'arrivée de signaux. Pour cela, nous allons utiliser les masques de signaux.

Les masques de signaux sont des masques qui contiennent un groupe de signal. Grâce à la fonction sigprocmask, nous pouvons bloquer ou débloquer l'arrivée des signaux définis par le masque donné en paramètres. La structure représentant le masque est opaque. Cela signifie qu'il ne faut pas remplir les

différents champs de la structure soi-même mais utiliser les fonctions mises à disposition. La foncction *sigemptyset* permet d'initialiser le masque reçu en paramètre. Les fonctions *sigaddset* et *sigdelset* permet d'ajouter des signaux aux masques. Toute tentative d'ajouter SIGKILL ou SIGSTOP sera tout simplement ignorée silencieusement par le système. Il existe d'autres fonctions associées aux masques que nous pouvons trouver à la page de manuel sigemptyset(3).

Récrivons notre programme en bloquant l'arrivée des signaux lors de l'entrée en zone critique.

Code

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

int diviseur;

void traitement(int signal)
{
    diviseur = 0;
    printf("Diviseur vaut désormais 0 \n");
}

int main()
{
    struct sigaction action;
    action.sa_handler = traitement;
    sigaction(SIGUSR1, &action, NULL);
    long int val = 10000000000;
    diviseur = 10;
    sigset_t masque;
    sigemptyset(&masque);
    sigaddset(&masque, SIGUSR1);
    if (fork() == 0)
    {
        sleep(4);
        kill(getppid(), SIGUSR1);
        printf("Signal SIGUSR1 envoyé \n");
        exit(0);
    }
    else
    {
        sigprocmask(SIG_BLOCK, &masque, NULL); //entrée en zone critique
        while (val != 100)
        {
            val = val / diviseur;
            printf("%ld \n", val);
            sleep(1);
        }
        sigprocmask(SIG_UNBLOCK, &masque, NULL); //fin de la zone critique
    }
```

```
42 }
```

Output

```
1000000000
100000000
10000000
1000000
Signal SIGUSR1 envoyé
100000
10000
1000
100
Diviseur vaut désormais 0
```

Nous constatons que le signal est bien envoyé pendant la zone critique mais que celui-ci est mis en attente. Le signal a été traité lorsque le signal était débloqué.

3 Limites et contraintes des signaux

3.1 Une bascule

Un dernier cas concret que nous allons réaliser est une bascule qui reçoit deux signaux différents. En fonction du signal reçu, la bascule se mettra à jour. La bascule a deux états possibles : UP et DOWN. Nous pourrions être tentés d'écrire un programme qui se met à jour en utilisant les 2 signaux classiques SIGUSR1 SIGUSR2 mis à la disposition du programmeur. Mais ce programme serait bien trop incomplet pour plusieurs raisons. Tout d'abord les signaux classiques n'ont pas de file d'attente. Si notre programme reçoit trop de signaux d'un coup, l'état de notre bascule à un moment T ne serait pas forcément cohérent par rapport à la liste de signaux envoyés car les signaux accumulés seraient tout simplement perdus.

Exemple de code

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5
6  int diviseur;
7
8  void traitement(int signal)
9  {
10     diviseur = 0;
11     printf("Diviseur vaut désormais 0 \n");
12 }
13
14 int main()
15 {
16     struct sigaction action;
```

```
17      action.sa_handler = traitement;
18      sigaction(SIGUSR1, &action, NULL);
19      long int val = 10000000000;
20      diviseur = 10;
21      sigset_t masque;
22      sigemptyset(&masque);
23      sigaddset(&masque, SIGUSR1);
24      if (fork() == 0)
25      {
26          sleep(4);
27          kill(getppid(), SIGUSR1);
28          printf("Signal SIGUSR1 envoyé \n");
29          exit(0);
30      }
31      else
32      {
33          sigprocmask(SIG_BLOCK, &masque, NULL); //entrée en zone critique
34          while (val != 100)
35          {
36              val = val / diviseur;
37              printf("%ld \n", val);
38              sleep(1);
39          }
40          sigprocmask(SIG_UNBLOCK, &masque, NULL); //fin de la zone critique
41      }
42  }
```

Output

```
Signal reçu
0
1
2
3
4
```

Nous pouvons observer que le premier signal est bloqué par le masque, mis dans la file d'attente et ensuite traité une fois masque levé. Les deux autres signaux envoyés n'ont jamais été traités et ont été perdus.

4 Les signaux temps-réel

4.1 Caractéristiques des signaux temps-réel

Pour écrire notre bascule, nous allons utiliser les signaux temps-réel qui, en plus d'ajouter des signaux supplémentaires pour le programmeur, ajoute une file d'attente des signaux reçus. Une autre caractéristique qui les différencie de ceux classiques des signaux temps-réel sont leur priorité. La documentation ne précise pas de priorité de traitement lorsque plusieurs signaux arrivent alors que les signaux temps-réel sont traités par odre croissant. Cela signifie que les signaux temps-réel

de plus petite valeur seront traités en priorité. Enfin, les signaux temps-réel possèdent une structure supplémentaire permettant de stocker une variable entière ou un pointeur. Cela peut être utilisé pour communiquer des informations entre deux processus.

4.2 Cas concret : une bascule

Les signaux temps-réel apporte donc tout ce dont nous avions besoin pour parvenir à coder cette bascule. Si nous gardons la même logique que nous avions imaginée plus tôt en utilisant deux signaux différents : un premier pour l'état UP et un second pour l'état DOWN. Cela donnerait un code ressemblant à ceci.

Code

```
1  #if defined(_POSIX_REALTIME_SIGNALS)
2  #error "Pas de signaux temps-réel disponibles"
3  #endif
4
5  #include <stdlib.h>
6  #include <signal.h>
7  #include <stdio.h>
8  #include <stdbool.h>
9  #include <unistd.h>
10
11 #define SIGRTUP (SIGRTMIN)
12 #define SIGRTDOWN (SIGRTMIN + 1)
13
14 bool etat = false;
15
16 void traitement(int signal, siginfo_t *info, void *inutile)
17 {
18     if (signal == SIGRTUP)
19         etat = true;
20     else
21         etat = false;
22     printf("Bascule en-état %d \n", etat);
23 }
24
25 int main()
26 {
27     struct sigaction action;
28     action.sa_flags = SA_SIGINFO;
29     action.sa_sigaction = traitement;
30     sigaction(SIGRTUP, &action, NULL);
31     sigaction(SIGRTDOWN, &action, NULL);
32     sigset_t masque;
33     sigemptyset(&masque);
34     sigaddset(&masque, SIGRTUP);
35     sigaddset(&masque, SIGRTDOWN);
36     sigprocmask(SIG_BLOCK, &masque, NULL);
37     for (int i = 0; i < 10; i++)
38     {
```

```
        raise(SIGRTUP);
        printf("Signal SIGRTUP envoyé \n");
        raise(SIGRTDOWN);
        printf("Signal SIGRTDOWN envoyé \n");
    }
    sigprocmask(SIG_UNBLOCK, &masque, NULL);
}
```

Output

```
Signal SIGRTUP envoyé
Signal SIGRTDOWN envoyé
Signal SIGRTUP envoyé
Signal SIGRTDOWN envoyé
Signal SIGRTUP envoyé
Signal SIGRTDOWN envoyé
Signal SIGRTUP envoyé
Signal SIGRTDOWN envoyé
Signal SIGRTUP envoyé
Signal SIGRTDOWN envoyé
Signal SIGRTUP envoyé
Signal SIGRTDOWN envoyé
Signal SIGRTUP envoyé
Signal SIGRTDOWN envoyé
Signal SIGRTUP envoyé
Signal SIGRTDOWN envoyé
Signal SIGRTUP envoyé
Signal SIGRTDOWN envoyé
Signal SIGRTUP envoyé
Signal SIGRTDOWN envoyé
Bascule en-état 0
Bascule en-état 0
Bascule en-état 0
Bascule en-état 0
Bascule en-état 0
Bascule en-état 0
Bascule en-état 0
Bascule en-état 0
Bascule en-état 0
Bascule en-état 0
Bascule en-état 1
Bascule en-état 1
Bascule en-état 1
Bascule en-état 1
Bascule en-état 1
Bascule en-état 1
Bascule en-état 1
Bascule en-état 1
Bascule en-état 1
Bascule en-état 1
```

Remarques

Nous bloquons l'arrivée des signaux afin de forcer notre bascule à gérer d'un coup une quantité de signaux d'un coup et d'illustrer la file d'attente. Nous pouvons donc observer que les signaux sont bien mis dans l'ordre d'arrivée dans la file d'attente mais un problème persiste. Ils ne sont pas traités selon l'ordre d'arrivée mais selon leur priorité. Cela est un problème car l'état de notre bascule ne correspond pas à l'arrivée des signaux. Pour y parvenir, nous devons donc utiliser un seul signal et pour pouvoir différencier un signal UP d'un signal DOWN, nous allons utiliser la dernière propriété de ces signaux en transmettant une valeur avec le signal.

Code

```
1  #if defined(_POSIX_REALTIME_SIGNALS)
2  #error "Pas de signaux temps-réel disponibles"
3  #endif
4
5  #include <stdlib.h>
6  #include <signal.h>
7  #include <stdio.h>
8  #include <stdbool.h>
9  #include <unistd.h>
10
11 #define SIGRTUPDATE (SIGRTMIN)
12
13 bool etat = false;
14
15 void traitement(int signal, siginfo_t *info, void *inutile)
16 {
17     etat = info->si_value.sival_int;
18     printf("Bascule en état %d \n", etat);
19 }
20
21 int main()
22 {
23     struct sigaction action;
24     action.sa_flags = SA_SIGINFO;
25     action.sa_sigaction = traitement;
26     sigaction(SIGRTUPDATE, &action, NULL);
27     if (fork() == 0)
28     {
29         union sigval valeur;
30         for (int i = 0; i < 10; ++i)
31         {
32             valeur.sival_int = 1;
33             sigqueue(getppid(), SIGRTUPDATE, valeur);
34             printf("Signal SIGRTUPDATE envoyé. Valeur = %d \n", valeur.sival_int);
35             valeur.sival_int = 0;
36             sigqueue(getppid(), SIGRTUPDATE, valeur);
37             printf("Signal SIGRTUPDATE envoyé. Valeur = %d \n", valeur.sival_int);
38         }
39         exit(0);
```

```
    }
    else
    {
        sleep(1);
    }
}
```

Output

```
Signal SIGRTUPDATE envoyé. Valeur = 1
Signal SIGRTUPDATE envoyé. Valeur = 0
Signal SIGRTUPDATE envoyé. Valeur = 1
Signal SIGRTUPDATE envoyé. Valeur = 0
Signal SIGRTUPDATE envoyé. Valeur = 1
Signal SIGRTUPDATE envoyé. Valeur = 0
Signal SIGRTUPDATE envoyé. Valeur = 1
Signal SIGRTUPDATE envoyé. Valeur = 0
Signal SIGRTUPDATE envoyé. Valeur = 1
Signal SIGRTUPDATE envoyé. Valeur = 0
Signal SIGRTUPDATE envoyé. Valeur = 1
Signal SIGRTUPDATE envoyé. Valeur = 0
Signal SIGRTUPDATE envoyé. Valeur = 1
Signal SIGRTUPDATE envoyé. Valeur = 0
Signal SIGRTUPDATE envoyé. Valeur = 1
Signal SIGRTUPDATE envoyé. Valeur = 0
Signal SIGRTUPDATE envoyé. Valeur = 1
Signal SIGRTUPDATE envoyé. Valeur = 0
Signal SIGRTUPDATE envoyé. Valeur = 1
Signal SIGRTUPDATE envoyé. Valeur = 0
Bascule en état 1
Bascule en état 0
Bascule en état 1
Bascule en état 0
Bascule en état 1
Bascule en état 0
Bascule en état 1
Bascule en état 0
Bascule en état 1
Bascule en état 0
Bascule en état 1
Bascule en état 0
Bascule en état 1
Bascule en état 0
Bascule en état 1
Bascule en état 0
Bascule en état 1
Bascule en état 0
Bascule en état 1
Bascule en état 0
```

Nous voyons cette fois-ci que notre dernier problème est résolu et que notre bascule fonctionne correctement.

5 Conclusion

Nous avons vu que les signaux classiques permettaient pas mal de possibilités mais que ceux-ci avaient leurs limites. Les signaux temps-réels les complètent mais modifient légèrement leur logique d'utilisation. Il faut aussi faire attention aux bugs que ceux-ci peuvent générer et savoir s'en prémunir.