Université de Technologie de Compiègne

SR02

Rapport de TD

2 - Appels système, création de processus et communication par signaux

Printemps 2015

Anaïs NEVEUX - Romain PELLERIN
Groupe 2
21 mars 2015

Table des matières

1 Exercices

1.1 Exercice 1

Le but de cet exercice est de gérer une prise de température régulière. Le thermomètre est simulé par un processus fils qui va générer une température aléatoire. Ce processus sera en attente du signal **SIGUSR1** et effectuera la prise de température à la réception. Concrètement, il s'agit ici de manipuler les gestionnaires de signaux de telle sorte à effectuer une action à la réception d'un signal.
Nous avons donc ici deux gestionnaires : l'un pour la prise de température, et l'un pour l'envoi de signal grâce à une alarme.

Le gestionnaire de l'alarme va réagir, comme son nom l'indique, au signal **SIGALRM**. Il effectuera un `kill(...)` sur le processus fils, du signal **SIGUSR1**, puis « remettra en route l'alarme ».

C

```c
...
#define DUREE 5

pid_t son;

void nouveau_gestionnaire(int num) {
    kill(son,SIGUSR1);
    alarm(DUREE);
}
...
```

Le gestionnaire du fils, lui, agit de la manière suivante : on génère une température aléatoire, puis on l'affiche.

C

```c
...
void signal_recu_par_le_fils(int num) {
    int r = rand();
    printf("Temperature mesuree : %ld degres\n",r%31 + 10); // entre 10 et 40
}
...
```

Vient donc le programme principal permettant la mise en place de notre programme. Le principe est simple :

1. On crée un fils puis on va ensuite séparer les différents traitements :
   - Dans le cas d'une erreur lors de la création du fils, nous nous contentons d'afficher un message d'erreur.
   - Dans le cas où la valeur retournée est 0, il s'agit du traitement concernant le processus fils. Dans cette partie nous créons un `struct sigaction` dont nous modifions l'action

exécutée, ici nous lui passons donc l'adresse du gestionnaire du fils. La fonction sigaction permet le remplacement du gestionnaire par défaut par le nouveau gestionnaire, et ce pour le signal **SIGUSR1**. Le programme se met ensuite dans une boucle pause.

— Dans les autres cas, nous sommes dans le processus père, qui lui aussi va installer un gestionnaire : celui concernant le signal **SIGALRM**. L'alarme est ensuite enclenchée et le processus entre dans une boucle infinie se contentant de faire un sleep d'une seconde puis d'afficher un '-' pour simuler une barre de progression.

1.2 Exercice 2

Ici il s'agissait d'alterner entre processus père et processus fils dans l'optique d'afficher des caractères en majuscule et miniscule de manière « incrémentale », c'est-à-dire que lors de la première exécution du handler des processus, ceux-ci vont afficher un seul caractère, puis deux, puis trois, et ainsi de suite, jusqu'à ce que toutes les lettres soient affichées.

Plusieurs solutions sont possible afin d'arriver à ce résultat, mais la structure générale et le programme principal reste globalement les mêmes.

— Une solution utilise directement les valeurs suivant la dernière lettre de chaque série, c'est-à-dire b, d, g, k, p, v et le caractère suivant z dans la table ascii. Ces valeurs permettent de savoir quand envoyer un signal au fils ou au père.

— Une autre solution serait d'incrémenter une variable correspondant au nombre maximum de lettres que le processus devra afficher à « chaque tour ».

C'est cette dernière méthode qui a été conservée dans le cadre de ce TD.

De la même manière que précédemment, nous aurons deux processus avec chacun un gestionnaire propre (même fonction). A chaque tour un nombre précis de lettre est affichée, jusqu'à avoir atteint un caractère de fin (soit 'Z' soit 'z'), représenté par la variable `end_char`.

C

```c
void signal_recu(int num) {
    int i;
    for (i = 0; i < counter; i++) {
        printf("%c", current_char++);
        fflush(stdout); // Pour afficher
        if (current_char > end_char)
            break;
    }
    kill(other,SIGUSR1);
    if (current_char > end_char)
        _exit(0);
    counter++; // Vaut initialement 1 au début du programme
}
```

Le programme principal se présente comme suit :

1. Fork
2. Récupération du pid du processus père par le fils et récupération du pid du processus fils par le père.
   (a) Le processus fils initialise ses variables (caractère de début et fin) et crée un `struct sigaction`, dont le handler associé sera une fonction générique. On envoie ensuite un message à soi-même afin d'initialiser le programme. Le processus rentre ensuite dans une boucle infinie et attend un signal.

(b) Le processus père initialise ses variables (caractère de début et fin) et va lui aussi créer un `struct sigaction` dont le handler pointe sur la même fonction. Ce processus entre ensuite dans une boucle infinie et attend un signal.

Ci-dessous le code associé :

C

```
...
int main() {
    switch (other = fork()) { // PID du fils
        case 0:
            //printf("Fils créé\n");
            other = getppid(); // PID du père
            current_char = 'a';
            end_char = 'z';
            break;
        case -1:
            printf("Erreur\n");
            return 1;
        default:
            //printf("Père créé.\n");
            current_char = 'A';
            end_char = 'Z';
    }
    struct sigaction recu;
    recu.sa_handler = &signal_recu;
    sigaction(SIGUSR1, &recu, NULL);
    if (current_char == 'a')
        kill(getpid(),SIGUSR1); // Fils s'auto lance
    while(1)
        pause();
    return 0;
}
```

1.3 Exercice 3

Cette exercice introduit les fonctions `setjmp(jmp_buf jb)` et `longjmp(jmp_buf jb,int i)` qui permettent respectivement de sauvegarder le contexte et de le recharger. Nous allons voir ici comment repérer un accès mémoire non autorisé, grâce aux gestionnaires de signaux, et à ces deux fonctions.

Deux fonctions ont été définies : la fonction `probe()` et un `signal_recu()` (handler du signal **SIGSEGV**, signal envoyé lors d'un accès illicite à une adresse mémoire). Le handler ne se contente que d'une seule chose : restaurer le contexte sauvegardé dans la variable `jb`, et envoyer la valeur 1, afin d'indiquer dans le reste du code qu'une erreur est survenue. Ce handler n'est appelé qu'en cas de violation d'accès mémoire.

La fonction `probe()` que nous avons créée agit comme suit :

— Sauvegarde du contexte et affectation de la valeur de retour de `setjmp(jmp_buf jb)` à `status`.

— Si le `status` vaut 0 (pas de violation mémoire donc handler non appelé), pas d'erreur, donc on affiche la valeur stockée et on renvoie 1.

— Sinon on renvoie 0 (handler appelé, qui aura retourné la valeur 1).

Dans le cas où un accès illicite est détecté, le programme va donc nous afficher l'erreur, l'index et l'adresse.

C

```
...
void signal_recu(int num) { // appelé si violation mémoire
    longjmp(jb,1); // retourne l'execution après l'appel setjmp
}

int probe(int *addr) {
    int status;
    status = setjmp(jb); // sauvegarde le contexte de pile et d'environnement
    if (status == 0) {
        printf("%d\n",*addr);
        return 1;
    }
    else
        return 0;
}
...
```

1.4 Exercice 4

Cet exercice consiste en la captation de la combinaison Ctrl+C, qui envoie un signal **SIGINT**. De cette manière, lorsque l'on va utiliser cette combinaison sur le processus fils, l'interface graphique de ce dernier deviendra vert pendant 2 secondes. Lors d'un clic sur le bouton _0_ de l'interface graphique, un **SIGINT** sera envoyé au père. Dans les deux cas, les signaux **SIGINT** sont comptabilisés afin que les processus se terminent après la réception de 3 signaux **SIGINT**.

Afin de réaliser cet exercice nous avons tout d'abord créé un entier `NB_SIGINT`, global, qui nous servira de compteur. Nous avons aussi intégré un entier `n` afin d'indiquer le temps restant lors de la réception du signal.

Nous avons ici deux handlers : `captpere()` et `captfils()`.
`captpere()` commence par incrémenter `NB_SIGINT`, indique le temps restant puis, si `NB_SIGINT` $>$ 3, stoppe le processus père.
`captfils()`, quant à lui, rend le rectangle vert pendant 2 secondes, incrémente `NB_SIGINT` et si `NB_SIGINT` $>$ 3, tue le processus fils.

Le programme principal se présente de la manière suivante :

1. Récupération du pid du processus pere.
2. Création d'un `struct sigaction`, association du handler `captpere()` puis installation du gestionnaire
3. Création du fils puis filtrage
   (a) Le fils installe son propre gestionnaire, crée la fenêtre puis se met dans l'attente d'un clic. Si ce clic correspond à la touche _0_, alors un signal **SIGINT** est envoyé au père. Si ce clic est -1 alors le fils s'arrête.
   (b) Dans le cas d'une erreur un message d'erreur s'affiche
   (c) Le père se contente de se mettre dans une boucle infinie et d'effectuer un `sleep()` de 10 secondes.

La seconde partie de l'exercice est de créer un programme sign.c basé sur celui effectué précedemment. Les handlers ne changent pas par rapport à la partie précédente. Les adaptations se font dans le programme principal.

Tout d'abord nous avons remplacé les variables `pid_t pere, fils` par un tableau contenant les pid. La seule partie modifiée ensuite est le processus fils. En effet celui doit désormais créer un second fils qui va lui-même créer une interface graphique. Ces deux fils auront les actions suivantes :

- Si on appuie sur 0 : envoie d'un **SIGINT** au processus pere
- Si on appuie sur 1 : envoie d'un **SIGINT** au processus fils1
- Si on appuie sur 2 : envoie d'un **SIGINT** au processus fils2
- Si on appuie sur 3 : envoie d'un **SIGINT** à tous les processus

Nous n'avons pas pu tester notre code pour cet exercice 4 car le programme ne fonctionnait pas (ne compilait pas) sur nos machines personnelles, il manquait la bibliothèque /usr/X11R6/lib.