TD 3

Processus, signaux et pipes

EXERCICE 1 ▶ Gestionnaire de signaux

On vous demande d'écrire un pseudo-algorithme dans lequel le processus principal exécute de façons répétée l'une des deux procédures de travail, appelées `faire_des_trucs()` ou `faire_des_choses()` : ces deux procédures se terminent forcément en un temps fini. Initialement, le processus principal exécute de façon répétée `faire_des_trucs()`, mais lorsqu'il reçoit le signal `SIGUSR1`, il attend que la procédure en cours d'exécution se termine, puis passe à l'autre. De plus, quand le processus principal reçoit le signal `SIGINT`, il attend que la procédure en cours se termine, puis il se termine lui-même.

Vous devez mettre en place des gestionnaires pour les signaux `SIGUSR1` et `SIGINT` pour réaliser le comportement décrit ci-dessus. On rappelle qu'idéalement un gestionnaire de signal ne comporte pas de primitives systèmes, n'effectue pas d'entrées-sorties, et se termine le plus vite possible. Écrivez un pseudo-code pour votre programme.

1) Mettez en place un mécanisme pour gérer l'alternance entre les procédures `faire_des_trucs()` et `faire_des_choses()` lorsque le processus reçoit le signal `SIGUSR1`. Vous pouvez utiliser des variables et spécifier une procédure comme gestionnaire du signal `SIGUSR1`.

2) Faites la même chose pour gérer la terminaison du programme lorsque le signal `SIGINT` est reçu.

EXERCICE 2 ▶ Synchronisation à l'aide signaux

On ne vous demande pas d'écrire un code C/C++, mais du pseudo-code : restez au niveau algorithmique, en gardant à l'esprit ce que les primitives POSIX permettent de faire. On suppose que les appels aux primitives ne provoquent pas d'erreurs : il faudrait gérer les situations correspondantes lors d'une implantation en C/C++, mais ça n'est pas l'objet ici.

On vous demande d'écrire un programme dans lequel le processus principal doit créer un processus fils. Le père va afficher les entiers de 1 à 26 sur sa sortie standard, et le fils les lettres de `a` à `z`. Bien que les affichages soient effectués par deux processus distincts, on souhaite qu'ils se fassent en ordre alterné : 1, `a`, 2, `b`, 3, `c`, … De plus, le père doit se mettre en sommeil une seconde entre l'affichage de chaque entiers.

Proposer une solution pour mettre en place ce comportement, en utilisant le mécanisme des signaux; vous utiliserez pour cela les trois fonction suivantes (et quelques autres) :

— sleep(`t`) met le processus appelant en sommeil pour `t` secondes,

— kill(`pid`, `sig`) pour envoyer le signal `sig` au processus identifié par `pid`; il est suffisant ici d'utiliser le signal `SIGCONT`,

— pause_cont() met le processus appelant en attente de la réception d'un signal `SIGCONT`. Attention, contrairement à sleep() et kill(), pause_cont() n'est pas une primitive standard.

EXERCICE 3 ▶ Blocages impliquant des pipes

On considère le code ci-dessous, qui a déjà été rencontré en CM [1].

```
1  int main(void) {
2      int ret, p[2]; // p[0] lecture, p[1] écriture
3      pipe(p);
4
5      ret = fork();
6      if(ret > 0) { // processus père, écrivain
7          close(p[0]);
8          for(char c = 'a'; c <= 'z'; c++) {
9              cout << "(père) j'écris " << c << endl;
10             write(p[1], &c, 1);
11         }
12         close(p[1]);
13         waitpid(ret, NULL, 0);
14     }
15     else { // processus fils, lecteur
16         char c;
17         int i = 0;
18         close(p[1]);
19         while(read(p[0], &c, 1) == 1)
20             cout << "(fils) je lis " << c << endl;
21         close(p[0]);
22     }
23     return 0;
24 }
```

1. Pour le simplifier, les cas d'erreurs (`pipe()`, `fork()`, `write()`… peuvent renvoyer -1) sont ignorés dans ce code; dans une implémentation réelle, ils faudrait les prendre en compte pour rendre le code plus robuste et pouvoir le déboguer… Cette observation vaut aussi pour les autres codes du TD.

Avec l'inclusion des bons fichiers d'en-têtes, ce programme peut être compilé sans problème : commencez par résumer le comportement que l'on va observer à l'exécution.

Dans chacune des questions suivantes, on commente une ou plusieurs lignes du programme. Mais on suppose les questions sont indépendantes, et que seule la partie indiquée à la question considérée est commentée.

1) Si on commente le `close(p[1])` de la ligne 12, le programme ne se termine jamais : pourquoi?

2) Si on commente le `close(p[1])` de la ligne 18, le programme ne se termine jamais : pourquoi?

3) Que se passe-t-il à l'exécution si on commente le `close(p[0])` de la ligne 7, ou `close(p[0])` de la ligne 21?

4) Que se passe-t-il à l'exécution si on commente tout le code du fils (des lignes 18 à 21)?