Processus & tubes

Gabriel Hondet
gabriel.hondet@lsv.fr

29 novembre 2020

Exercice 1 : Fichiers louches.

Téléchargez le fichier `http://www.lsv.fr/~hondet/resources/archos/touch_weird.sh.x` et exécutez le. Trois fichiers (vides) devraient être apparus dans votre répertoire courant. Supprimez-les en utilisant *uniquement* la ligne de commande.
Indication : comment sont identifiés les fichiers de manière générale ?

Solution : Utiliser les inodes : `ls -i` et `find . -inum ... -exec rm {} \;`

Exercice 2 : Programme erroné.

Considérons `http://www.lsv.fr/~hondet/resources/archos/closed_pipe.c`. Le fils est censé imprimer les caractères que lui envoie le père : expliquez le dysfonctionnement et corrigez le programme.

Solution : The read end `p[0]` of the pipe is closed before forking, hence the child cannot read from it. Moving `close(p[0])` below `if (fork()) {` solves the problem.

Exercice 3 : Tubes et remplacement de code.

Écrivez un programme (en C) qui télécharge l'archive `http://www.lsv.fr/~hondet/resources/archos/shell-bootstrap.tar.gz`, et la décompresse sans créer de fichier temporaire. Dit autrement, on veut coder la commande `curl <url> | tar xz` en C. Les programmes `curl` et `tar` seront appelés par `exec` ou dérivés.

Solution : To perform the pipe, the main program forks into two sub programmes, one for each part of the pipe (*Question for the reader :* can we use only one fork ?). The main difficulty is to modify adequately file descriptors so that the download child writes the data where the decompressing child reads.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int p[2];
    if (pipe(p) == -1) perror("Pipe creation failed");
```

```c
    pid_t downloader = fork();
    if (downloader == 0) {
        close(p[0]);
        // We do not have to read from the pipe, so we close the read end
        dup2(p[1], STDOUT_FILENO);
        /* Replace the file descriptor STDOUT_FILENO with the file descriptor
         * p[1]. Hence, STDOUT_FILENO is linked to the write end of the pipe.
         * This way, any function that use STDOUT_FILENO, like printf, writes
         * the data to the pipe (instead of the terminal, by default). */
        close(p[1]);
        /* Since the file descriptor p[1] has been copied with dup2, the
         * write-end of the pipe has two file descriptors: p[1] and
         * STDOUT_FILENO. We close the one that won't be used, p[1]. */
        execlp("curl", "curl",
               "http://www.lsv.fr/~hondet/resources/archos/shell-bootstrap.tar.gz",
               NULL);
        /* execlp replaces the current image of the programme with another one.
         * In that case, the other programme is curl. The replacing programme
         * inherits its file descriptors from the parent process. In that case,
         * the STDOUT_FILENO of curl will hence be the write end of the created
         * pipe. */
    }
    pid_t untarer = fork();
    if (untarer == 0) {
        /* The logic is close to the one of the download process, except that
         * this process reads on the pipe. */
        close(p[1]);
        // We don't have to write on the pipe here.
        dup2(p[0], STDIN_FILENO);
        /* Copy read end of the pipe and replace stdin, so that subsequent
         * programmes read on the pipe. */
        close(p[0]);
        // Close the duplicated file descriptor
        execlp("tar", "tar", "xz", NULL);
    }
    return 0;
    /* The parent process doesn't have to wait for the termination of its
       children. The pipe created will exist as long as there are file
       descriptors open on one of its ends. */
}
```

Exercice 4 : La fonction d'Hénon, partie I.

On va calculer l'orbite d'un système dynamique de dimension 2. La fonction d'Hénon est définie par le système

$$H_{a,b} = \begin{cases} x_{n+1} = a - by_n - x_n^2 \\ y_{n+1} = x_n. \end{cases} \tag{1}$$

On utilisera un processus pour calculer la suite $(x_n)_n$ et un autre processus pour calculer la suite $(y_n)_n$. Les processus échangeront leurs données via un (ou des) tube(s). Pour éviter d'avoir à mettre en place une synchronisation entre les processus, on pourra utiliser des pauses `sleep(1)`.

Solution :

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>

#define _bound 400
#define _a 1.25
#define _b 0.3
#define _x0 0.5
#define _y0 0.5

void gnuplot_print(int fd, double x, double y) {
    char c = ' ';
    write(fd, &x, sizeof(double));
    write(fd, &c, sizeof(char));
    write(fd, &y, sizeof(double));
    c = '\n';
    write(fd, &c, sizeof(char));
}

void c_xn(int bound, int p[], int q[]) {
    close(p[1]);
    close(q[0]);
    double xn = _x0, yn = _y0;
    for (int i = 1; i < bound; i++) {
        read(p[0], &yn, sizeof(double));
        xn = 1 - _a * xn * xn + yn;
        printf("x%d: %lf\n", i, xn);
        write(q[1], &xn, sizeof(double));
    }
    exit(0);
}

void c_yn(int bound, int p[], int q[]) {
    close(p[0]);
    close(q[1]);
    double xn = _x0, yn = _y0;
    // Start computing y1
    yn = _b * xn;
    write(p[1], &yn, sizeof(double));
    for (int i = 2; i < bound; i++) {
        sleep(1);
        read(q[0], &xn, sizeof(double));
        yn = _b * xn;
        printf("y%d: %lf\n", i, yn);
        write(p[1], &yn, sizeof(double));
```

```
    }
    exit(0);
}

int main() {
    int p[2];
    int q[2];
    if (pipe(p) == -1) perror("p pipe");
    if (pipe(q) == -1) perror("q pipe");
    // open("henon.plain", O_CREAT | O_WRONLY);
    if (!(fork())) c_xn(_bound, p, q);
    if (!(fork())) c_yn(_bound, p, q);
    wait(NULL);
    wait(NULL);
}
```

Comment mettre en place une synchronisation par signaux ?

Solution : Le père commence par transmettre le *pid* des enfants entre eux, pour qu'ils puissent s'échanger des signaux. Ensuite on peut utiliser un `sigwait` bien placé.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <signal.h>

// Pipes and signals for synchronisation

#define _bound 100
double _a = 1.4;
double _b = -0.3;
double _x0 = 0.5;
double _y0 = 0.5;

sigset_t mask;

void c_xn(int bound, int p[], int q[]) {
    close(p[1]);
    close(q[0]);
    double xp = _x0, yp = _y0;
    double xn, yn;
    /* Reading the pid of the process computing yn */
    pid_t y_pid;
    /* We don't really need this variable, but sigwait expects a pointer. */
    int recvd;
    if (!(read(p[0], &y_pid, sizeof(pid_t)))) perror("reading yn pid");
    for (int i = 1; i < bound; i++) {
```

```
        // Wait for it...
        sigwait(&mask, &recvd);
        // Get data
        read(p[0], &yn, sizeof(double));
        xn = _a - _b * yp - xp * xp;
        // Send data
        write(q[1], &xn, sizeof(double));
        // Signal a data has been sent
        kill(y_pid, SIGUSR1);
        // Update everything
        xp = xn, yp = yn;
    }
    exit(0);
}

void c_yn(int bound, int p[], int q[]) {
    close(p[0]);
    close(q[1]);
    double xp = _x0, yp = _y0, xn, yn;
    pid_t x_pid;
    int recvd;
    if (!(read(q[0], &x_pid, sizeof(pid_t)))) perror("reading xn pid");
    // Start computing y1
    yn = xp;
    write(p[1], &yn, sizeof(double));
    kill(x_pid, SIGUSR1);
    yp = yn;
    for (int i = 2; i < bound; i++) {
        sigwait(&mask, &recvd);
        read(q[0], &xn, sizeof(double));
        // Print to standard out here (why not)
        printf("%lf %lf\n", xp, yp);
        yn = xp;
        write(p[1], &yn, sizeof(double));
        kill(x_pid, SIGUSR1); // Anounce there is a value to read
        xp = xn, yp = yn;
    }
    exit(0);
}

int main() {
    /* Setting up signals */
    sigemptyset(&mask);
    sigaddset(&mask, SIGUSR1);
    sigprocmask(SIG_BLOCK, &mask, NULL);

    int p[2];
    int q[2];
    if (pipe(p) == -1) perror("p pipe");
    if (pipe(q) == -1) perror("q pipe");
    pid_t xproc, yproc;
```

```
    if (!(xproc = fork())) c_xn(_bound, p, q);
    if (!(yproc = fork())) c_yn(_bound, p, q);
    /* Sending pids to the processes so that they can communicate with each
     * other. */
    write(q[1], &xproc, sizeof(pid_t));
    write(p[1], &yproc, sizeof(pid_t));
    close(p[1]); close(q[1]); close(p[0]); close(q[0]);
    wait(&xproc);
    wait(&yproc);
    return 0;
}
```

Par la suite, on créera en plus un processus dédié à la sortie : ce processus doit écrire des lignes sous la forme `0.3415 1.2451` où le premier nombre est x_n et le deuxième y_n dans un fichier `henon.dat`.

On pourra tracer la fonction avec la commande `gnuplot henon.p` après avoir téléchargé le script `http://www.lsv.fr/~hondet/resources/archos/henon.p`. Le fichier `henon.dat` doit être dans le même dossier que `henon.p`.

Pour $a = 1\ .4$ et $b = -0\ .3$, la fonction est chaotique (au sens de Devaney), c'est-à-dire,

- la fonction est sensiblement dépendante des conditions initiales (effet papillon, imprédictabilité) ;
- la fonction est topologiquement transitive (indécomposabilité) ;
- les points périodiques sont denses dans le domaine de définition de $H_{a,b}$ (régularité).

et possède un attracteur étrange.

Pour plus d'informations sur les systèmes dynamiques, voir *An introduction to Chaotic Dynamical Systems*, Robert L. Devaney, Westview.

Solution :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <time.h>
#include <string.h>

#define _a 1.4
#define _b (-0.3)
#define _x0 0.5
#define _y0 0.5

int bound = 100;

void c_xn(int p[], int q[], int wp[]) {
    close(p[1]);
    close(q[0]);
```

```
    close(wp[0]);
    double xn = _x0, yn = _y0;
    for (int i = 1; i < bound; i++) {
        xn = _a - _b * yn - xn * xn;
        write(q[1], &xn, sizeof(double));
        read(p[0], &yn, sizeof(double));
        write(wp[1], &xn, sizeof(double));
    }
    exit(0);
}

void c_yn(int p[], int q[], int wp[]) {
    close(p[0]);
    close(q[1]);
    close(wp[0]);
    double xn = _x0, yn = _y0;
    for (int i = 1; i < bound; i++) {
        sleep(1);
        printf("\r%d", i);
        fflush(stdout);
        yn = xn;
        // Sending to colleague
        write(p[1], &yn, sizeof(double));
        // Update x
        read(q[0], &xn, sizeof(double));
        // Sending to writer
        write(wp[1], &yn, sizeof(double));
    }
    exit(0);
}

void writer(int px[], int py[]) {
    close(px[1]);
    close(py[1]);
    double xn, yn;
    int fd;
    if ((fd = open("henon.dat", O_DSYNC | O_CREAT | O_RDWR | S_IRUSR)) == -1)
        perror("henon.dat");
    char buf[20];
    while (read(px[0], &xn, sizeof(double)) &
           read(py[0], &yn, sizeof(double))) {
        printf("%+0.4lf %+0.4lf\n", xn, yn);
        sprintf(buf, "%lf %lf\n", xn, yn);
        if (write(fd, buf, strlen(buf)) == -1) perror("write");
    }
    close(fd);
    exit(0);
}

int main() {
    int px[2], py[2];
```

```
    pipe(px);
    pipe(py);
    int p[2], q[2];
    if (pipe(p) == -1) perror("p pipe");
    if (pipe(q) == -1) perror("q pipe");
    if (!fork()) writer(px, py);
    if (!fork()) c_xn(p, q, px);
    if (!fork()) c_yn(p, q, py);
    wait(NULL);
    wait(NULL);
    // Close writers so that writer receives a SIGPIPE
    close(px[1]);
    close(py[1]);
    wait(NULL);
    exit(0);
}
```

Exercice 5 : Coquille vide.

Vous trouverez le squelette de base du code C que l'on va utiliser pour recoder un shell. Pour compiler le projet, utilisez la commande `make`. Par défault, le shell ne peut pas faire grand chose. On va essayer de le compléter pas à pas. On va d'abord s'intéresser à la fonction `execute` :

Le cas de base, correspond au cas `C_PLAIN`. Donner un exemple de commande qui une fois parsée retourne un objet `cmd` tel que `cmd->type == C_PLAIN`. En utilisant `ps`, observer ce qui se passe dans un terminal lorsque vous lancez une commande.

Pour le moment, toute commande de base sera tout simplement exécutée. Pour exécuter une commande, la librairie `glibc` offre un panel de fonctions dont on peut avoir un aperçu en utilisant la commande

```
man exec
```

Selon vous, quelle fonction serait la plus appropriée dans notre cas (justifier) ? En utilisant toutes ces observations, remplir le trou `C_PLAIN`.

Quel est le symbol pour l'opérateur de séquence en bash ? Donner un exemple de commande où la séquence se comporte différemment de l'opérateur *et* logique.

Implémenter le cas `C_SEQ`.

Implémenter les cas `C_AND` et `C_OR`.

Il est possible en bash d'écrire une commande de la forme (cmd1 && cmd2 | cmd3 ...) 2>/dev/null

Quel est le rôle des parenthèses dans la commande ci-dessus ? Donner un exemple d'une commande qui utilise (de façon non triviale) ces parenthèses. Implémenter le cas `C_VOID`.

Que se passe-t-il si vous faites `CTRL+C` dans notre shell ? Proposer une solution pour récupérer la main après que l'utilisateur a entré `CTRL+C`.

Que se passe-t-il lorsque que vous lancez la commande

```
ls > dump
```

Pour corriger ce problème, je vous invite à lire les pages de manuel `man stdin` et `man dup`. En utilisant toutes ces informations, implémenter la fonction `apply_redirections` puis modifier votre implémentation pour que la commande ci-dessus se comporte comme prévu.

Il nous reste finalement à implémenter le cas `C_PIPE`, pour cela je vous invite à regarder le manuel de `man pipe`. Donner un exemple qui met en évidence pourquoi on ne peut pas simplement utiliser `dup2` pour réimplémenter le pipe ? En utilisant la fonction `pipe`, réimplémenter le cas `C_PIPE`.

À ce stade, nous avons implémenté un *shell* très rudimentaire, cependant il est possible de l'étendre de bien des manières. Voici quelques possibilités d'extensions qui peuvent vous rapporter des points bonus :

- Réimplémenter les commandes `ls`, `cat` ou `cd`
- Implémenter l'extension des wildcards : `ls *.pdf`
- Implémenter les processus de fond via les commandes : `jobs`, `bg`, `fg`, …