Séances machine 1-2

Table des matières

1 Introduction à la programmation système

1.1 À quoi sert un système d'exploitation ?

— Faire tourner des applications.
— Exploiter les ressources d'une machine
— Ressources vue sous forme d'abstractions : "fichiers", "processus", "connexion réseau", etc.

API système : **Application Programming Interface**

— Bibliothèque d'**appels systèmes** utilisable pour la programmation d'applications

1.2 Objectifs, Approche

— connaissance des appels systèmes fondamentaux
— exemples pratiques
— programmation en C

Configuration Visual Studio Code C/C++

```
# C/C++ for Visual Studio Code (IntelliSense, code browsing, debugging, ...)
code --install-extension ms-vscode.cpptools
```

Options de compilation

```
gcc -std=c11 -Wall -Wextra -pedantic -D_XOPEN_SOURCE=700
```

1.3 Écriture en C

```
int n = 123;
fprintf( stdout, "hello, world %dn", n );
```

— formattage des données, remplissage d'un tampon avec la chaine `"hello, world 123n"`
— expédition sur la sortie standard

```
write(STDOUT_FILENO, tampon, 17);
```

1.4 Programme complet (taper, essayer)

```
#include <unistd.h>

int main()
{
    char tampon [ ] = "Hello, world 123n";
    write(STDOUT_FILENO, tampon, 17);
    return 0;
}
```

Paramètres :

— numéro de *descripteur*
— *adresse des données*
— le *nombre d'octets* à transférer.

1.5 Numéros de descripteurs

— Correspondent aux fichiers ouverts
  — 0 = `STDIN_FILENO` : entrée standard (`stdin`)
  — 1 = `STDOUT_FILENO` : sortie standard (`stdout`)
  — 2 = `STDERR_FILENO` : sortie d'erreur (`stderr`)
— Servent à les identifier dans les appels systèmes `read`, `write`, `close` ...

1.6 Exercice

— Utilisez `read()` pour lire une ligne dans un tableau de caractères
— `read()` retourne le nombre de caractères lus
— faire écrire ce qui a été lu.

Documentation :

```
man 2 read
```

2 Projet : un shell en C

Un shell est un programme interactif qui

— lit une ligne de commande
— l'analyse
— la fait exécuter
— recommence

2.1 Analyse d'une commande

Découper une ligne de commande `"cp abc/def /tmp"` en mots :

```
struct StringVector {
    size_t capacity;
    size_t size;
    char ** strings;  // dynamic array of pointers
};

void string_vector_init(struct StringVector * this, size_t capacity);
void string_vector_free(struct StringVector * this);

void string_vector_add(struct StringVector * this,
        const char *start,
        const char *end);
```

```
size_t string_vector_size(const struct StringVector * this);
char * string_vector_get(const struct StringVector * this,
        size_t index);

struct StringVector split_line(char * line)          <---
```

2.2 Travail 1 : écrire un programme qui gère la boucle

```
fini = faux
tant que pas fini
    lire une ligne
    la décomposer en mots
    selon le premier mot
    "exit"
        => fini = vrai
    "help"
        => afficher "tapez exit pour arrêter"
    autre
        => afficher "commande inconnue"
```

Amélioration possible : afficher un prompt, avec le numéro de commande qui s'incrémente.

2.3 Une structure possible pour un shell

```
struct Shell {
    bool running;
    int line_number;
    char * line;
    ssize_t line_length;
};

void shell_init(struct Shell *s);
void shell_run(struct Shell *s);
void shell_free(struct Shell *s);

void shell_read_line(struct Shell *s);
void shell_execute_line(struct Shell *s);

int main(int argc, char** argv)
{
    struct Shell shell;
    shell_init(& shell);
    shell_run(& shell);
    shell_free(& shell);
    return (EXIT_SUCCESS);
}
```

2.4 Travail 2 : sous-shell (1)

Faire reconnaitre la commande " !" qui lancera un "sous-shell" grâce à la commande

```
system("/bin/bash");
```

Amélioration :

— consultez (`getenv`) la variable d'environnement `SHELL` pour déterminer le shell à utiliser.

2.5 Travail 3 : commande interne `cd`

Faire reconnaitre la commande 'cd' qui change le répertoire courant

— appel à `chdir()`.
— Pour `cd` sans paramètre, utiliser la variable d'environnement `HOME`

2.6 Travail 4 : Lancement d'une commande dans un sous-shell (2)

Si la commande " !" a des paramètres, elle fait exécuter la ligne de commande par `system()`. Par exemple

```
!    ls  -l /tmp
```

lance

```
system("ls -l /tmp");
```

2.7 2 solutions

— utiliser la ligne courante lue par le shell (dans `line`) en "retirant" le caractère '!' ...
— écrire une fonction qui concatène les chaines d'un tableau (de chaine) en une chaine de caractères :

```
char *strjoinarray(char * dest, char *strings[], size_t number, char * glue) {

    size_t glue_length = strlen(glue);

    char *target = dest;              // where to copy the next elements
    *target = '0';
    for (size_t i = 0; i < number; i++) {
        if (i > 0) {                  // need glue ?
            strcat(target, glue);
            target += glue_length;
        }
        strcat(target, strings[i]);
        target += strlen(strings[i]);  // move to the end
    };
    return dest;
}
```

```c
int main() {
    char *tab[3] = {"!", "ls", "/tmp"};
    char *s = malloc( 256 * sizeof(char) );
    strjoinarray(s, tab, 3, ";");
    printf("%sn", s);                // !;ls;/tmp
    free(s);
    return 0;
}
```

Remettre à la séance suivante : un code commenté imprimé par étudiant(e).

2.8 Pointeurs de fonctions

Déclaration de type

```c
typedef void (*Action)();      // notation C
```

Affectation, appel

```c
void action_manger() {...}
void action_dormir( int duree ) {...}

Action a = action_manger;
a();

a = action_dormir;
a( 8 );
```

2.9 Restructuration du code

Actions = fonction

```c
#include "StringVector.h"

typedef void (*Action) (
        struct Shell *,
        const struct StringVector *
    );

Action get_action(char * name)
{
    int i = 0;
    while (actions[i].name != NULL
            && strcmp(actions[i].name, name) != 0) {
        i++;
```

```
    }
    return actions[i].action;
}
```

2.10 Restructuration (2)

Table d'actions

```
static struct {
    const char *name;
    Action action;
} actions[] = {
    { .name = "exit", .action = do_exit},
    { .name = "cd",   .action = do_cd},
    { .name = "pwd",  .action = do_pwd},
    { .name = "help", .action = do_help},
    { .name = "?",    .action = do_help},
    { .name = "!",    .action = do_system},
    { .name = NULL,   .action = do_execute}
};

static void do_help(struct Shell *this, const struct      StringVector *args)
{
    printf("-> commands: exit, cd, help, ?.n");
}
...
```

2.11 Restructuration (3)

```
struct StringVector tokens = split_line(this->line);
int nb_tokens = string_vector_size(& tokens);

if (nb_tokens == 0) {
    printf("-> Nothing to do !");
} else {
    char *name = string_vector_get(& tokens, 0);
    Action action = get_action(name);
    action(this, & tokens);

    string_vector_free(& tokens);
}
```

2.12 Travail à faire : restructuration

Restructurez votre code.

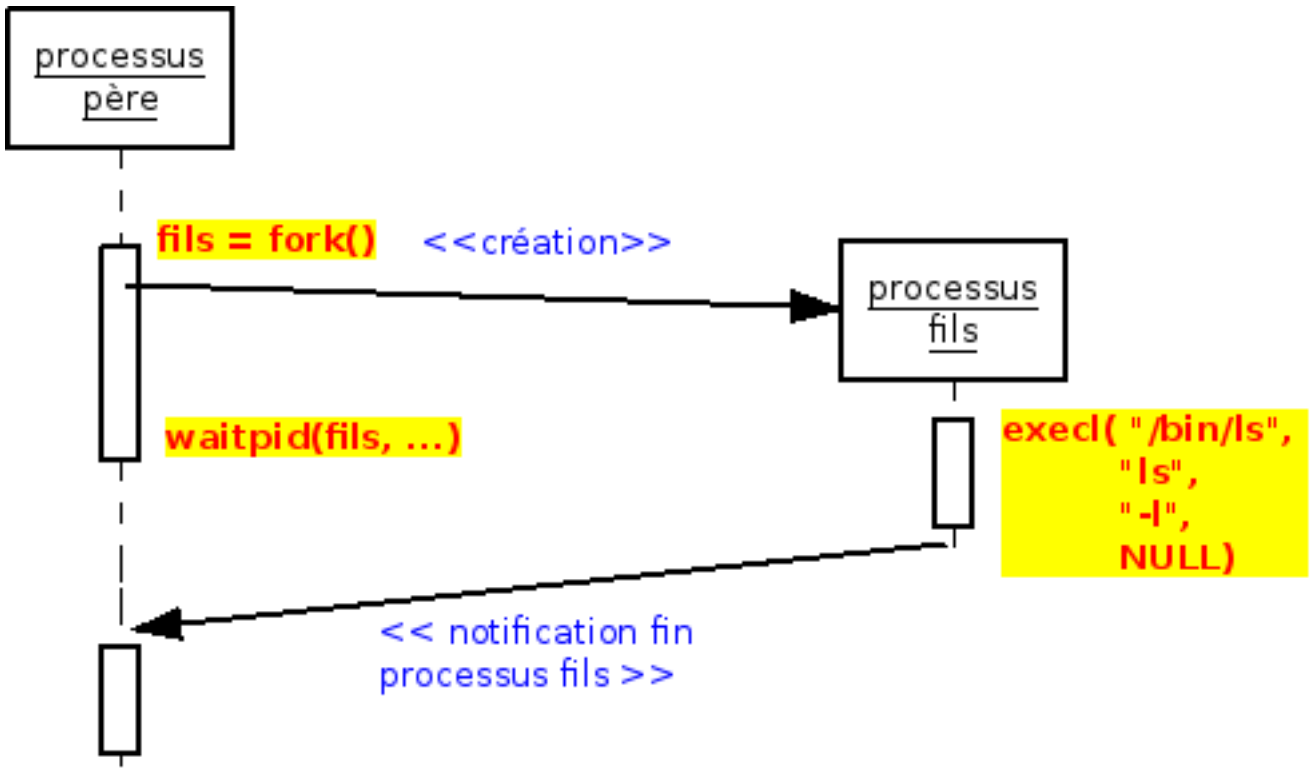

Figure 1 – Exécution de “ls -l”

2.13 Que fait `system()` ?

Fonction de bibliothèque, combine 3 appels système

- `fork()`, qui crée un nouveau processus
- `exec()`, qui exécute un fichier (exécutable)
- `waitpid()`, qui attend la fin d’un processus

2.14 Déroulement de `system("une commande")`

```
1. créer un nouveau processus (fils)

2. le processus fils
    - cherche le fichier exécutable
    - le copie dans son espace mémoire
    - lance son exécution

3. le processus père
    - attend que le fils se termine
```

2.15 Illustration

2.16 L’appel `fork()`

- Demande au système de créer un nouveau processus (fils)

— copie presque identique du processus appelant (père) :
  — même contenu de la mémoire,
  — mêmes fichiers ouverts, etc.

Différence : la fonction retourne

— 0 au processus fils
— le numéro du fils au père

Note : Il se peut aussi que le `fork()` échoue (retourne -1 au père).

2.17 Illustration

```
pid_t p = fork();
if (p == 0) {
    printf("je suis le processus filsn");
    ...
    exit (EXIT_SUCCESS);
}

printf("je suis le processus pèren");
printf("le processus fils a le numéro %dn",p);
....
}
```

2.18 Travail avec `fork()`

Ecrire un programme C++ qui

— affiche 5 fois “tip”, avec un délai (`sleep`) de 3 secondes,
— après avoir lancé un processus fils qui affiche 10 fois “top” avec un délai de 2 secondes.

2.19 Application de `fork()`

Ajoutez au “shell” une commande qui affichera un message de rappel dans un délai indiqué (en secondes)

```
> rappel 30 aller manger

...
RAPPEL : aller manger
```

Remarquez (ps) l’apparition de *zombies*.

2.20 L’appel système `wait()` / `waitpid()`

La fonction `waitpid()`

— attend qu’un processus fils se termine,
— récupère un `int` qui combine plusieurs informations sur l’exécution du fils. `WEXITSTATUS` extrait le code de retour

```
    pid_t fils = fork();
    ...
    int status;
    waitpid (fils, &status, 0);     // passage par adresse
    printf( "le processus %d s'est terminé avec le code de retour %dn",
        fils,
        WEXITSTATUS(status) );
```

2.21 `wait()`

Le troisième paramètre est une combinaison d'options. Voir la doc.

Il existe également un appel `wait` qui permet d'attendre un processus fils non spécifié. Il équivaut à `waitpid(-1, &status, 0)`.

2.22 Exercice `fork + wait` : la course de haies

On simule une course de haies 4x100 m entre 6 équipes.

Chaque équipe est simulée par un processus, avec tirage aléatoire de la durée.

```
pour j de 1 à 4
| afficher "le coureur j de l'équipe n est parti"
| attendre de 8 à 11 secondes
afficher "l'équipe n est arrivée"
```

2.22.1 Version 1

Dans un premier temps, les équipes sont identifiées par leur numéro de processus.

Le `wait()` fera afficher les équipes avec leur rang :

```
1. équipe #1234
2. équipe #1236
...
```

Éléments techniques nécessaires

- appel `getpid()` pour connaitre le numéro d'un processus
- Génération de nombres aléatoires entiers entre a et b :

```
srand(time());
...
int r = a+ rand() % (b-a+1);
```

2.22.2 Version 2 : chutes

Chaque coureur a une chance sur 10 de **tomber**.

- Dans ce cas, le processus de l'équipe se termine avec `EXIT_FAILURE`.
- Et l'équipe ne figure pas dans la liste finale.

2.22.3 Version 3 : équipes avec noms

Le programme prend en paramètre les noms des équipes (paramètres 1 à `argc-1` de l' `argv` du `main`) :

```
$ course FRA USA ITA CHN
- le coureur 1 de FRA est parti
- le coureur 1 de USA est parti...

1. FRA
2. CHN
...
```

une table de correspondance permettra d'afficher le nom à partir du numéro de processus retourné par `wait()`.