Séances machine 1-2

S3 - M3101

2017-2017

(Semaine 37) Présentation du cours

À quoi sert un système d'exploitation ?

- Exploiter les ressources d'une machine
- Faire tourner des applications.
- 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

Objectifs, Approche

- connaissance des appels systèmes fondamentaux
- exemples pratiques
- programmation en C++

Configuration QT Creator

- Configuration .pro pour avoir la compatibilité avec les standards langage C++11, conformité avec POSIX et extensions OpenGroup 7

```
QMAKE_CXXFLAGS = -std=c++11
QMAKE_CXXFLAGS += -Wall -Wextra -pedantic

QMAKE_CXXFLAGS += -D_POSIX_C_SOURCE=200809L
QMAKE_CXXFLAGS += -D_XOPEN_SOURCE=700
```

Exemple

Écriture en C++

```
int n = 123;
cout << "hello, world " << endl ;
```

Ce qui se passe

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

```
write(STDOUT_FILENO, tampon, 18);
```

Programme complet (taper, essayer)

```c
#include <unistd.h>

int main()
{
    char tampon [ ] = "Hello, world 123\n";
    write(STDOUT_FILENO, tampon, 18);
    return 0;
}
```

Paramètres :

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

Numéros de descripteurs

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

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.

Utiliser les appels système en C++

Conventions d'appel

- Les appels système sont des appels C.
- il va falloir adapter

Les chaines de caractères :

- en C : tableau (pointeur) de caractères, terminé par un caractère nul.
- en C++ : `string` = vecteur de caractères

Adaptation : de string à chaine “C”

```
// Code C++
#include <unistd.h>

#include <iostream>
using namespace std;

int main()
{
  string chaine = "bonjour";
  const char * s = chaine.c_str();
  int l = chaine.size();
  write(STDOUT_FILENO, s, l);
  return 0;
}
```

Adaptation : de chaine “C” à string :

Constructeurs de `string`

```
default
        string();
copy
        string (const string& str);
from c-string
        string (const char* s);
from buffer
        string (const char* s, size_t n);
```

Projet : un shell en C++

Programme interactif qui

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

Best of two worlds

- C++ parce qu'on a les chaines, les conteneurs, les classes, etc.
- besoin de faire des appels systeme

Intérêt de C++ dans le projet

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

```
vector<string> decouper(const string &ligne)
{
  vector<string> mots;
  istringstream in (ligne);
  string m;
    while (in >> m) {
        mots.push_back(m);
    }
    return mots;
}
```

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.

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.

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’environnment `HOME`

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

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

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

lance

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

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

(Semaine 38) Restructuration du code

Actions = fonction

```
using Action = void (*)();      // déclaration de type

bool encore;

void action_manger() {
  cout << "miam" << endl;
}

void action_partir() {
  encore = false;
  cout << "zzzz" << endl;
}
```

restructuration (2)

Table d'actions

```
const map<string, Action> actions {
    { "manger",   action_manger },
    { "dormir",   action_dormir },
    { "partir",   action_partir },
    ...
    };
```

En fait, une map.

restructuration (3)

```
int main()
{
  encore = true;
  while (encore) {
    string mot;
    cout <<  "> ";
    cin >> mot;
    auto it = actions.find(mot);    // recherche
    if (it == actions.end()) {
        cout << "Commande '" << mot
             << "' inconnue" << endl;
    } else {
      it->second ();                // appel
    }
  }
```

Pointeurs de fonctions

Déclaration de type

```
using Action = void (*)();      // déclaration de type
typedef void (*Action)();       // notation C
```

Affectation, appel

```
Action a = manger;
....
a();
```

Travail à faire : restructuration

Restructurez votre code.

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

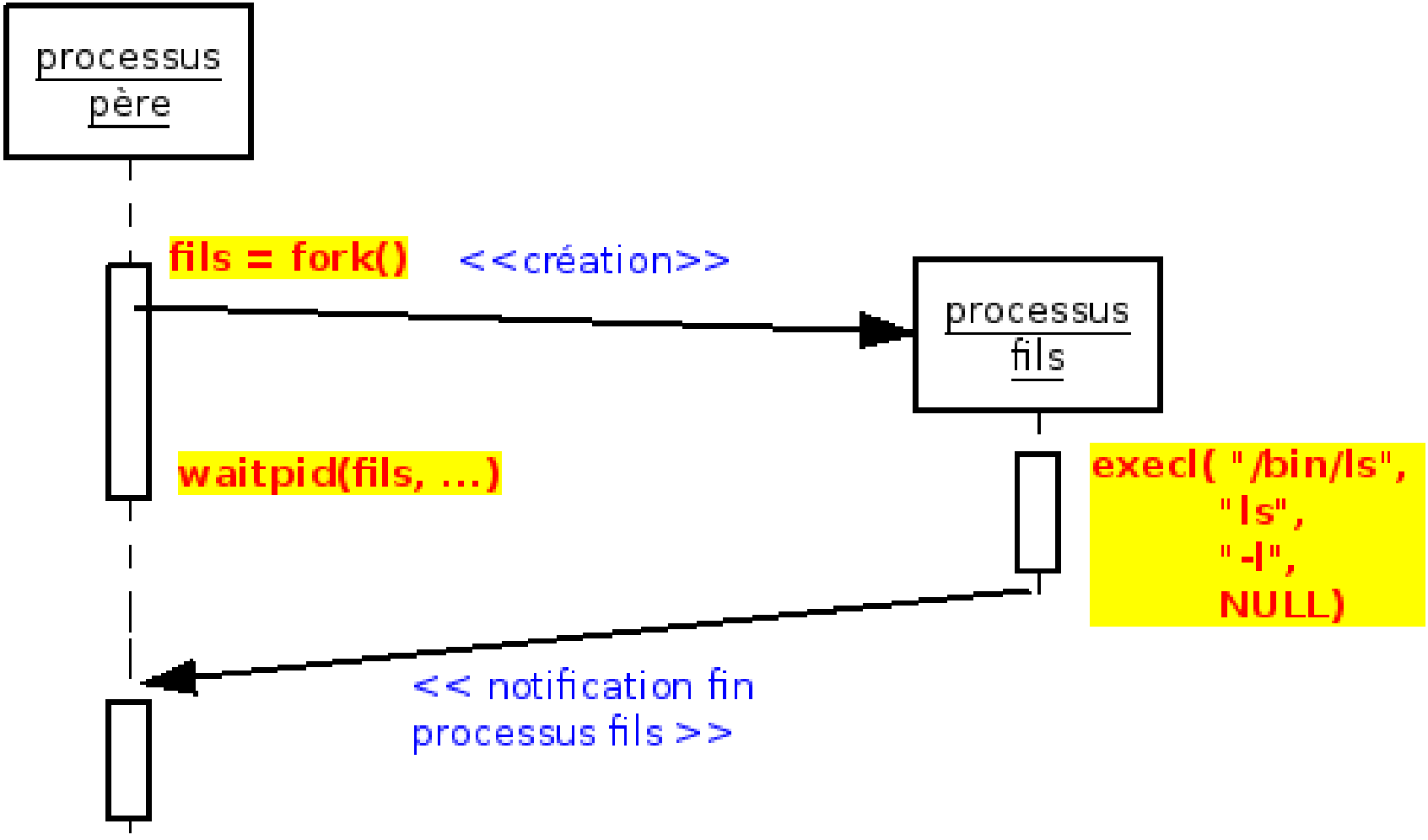
Déroulement de `system("une commande")`

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

Illustration :



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).

Illustration

```
pid_t p = fork();
if (pid_t == 0) {
    cout << "je suis le processus fils " << endl;
    ...
    exit (EXIT_SUCCESS);
}

cout << "je suis le processus père" << endl;
cout << "le processus fils a le numéro " << p << endl;
....
}
```

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.

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*.

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
cout << "le processus " << fils
     << " s'est terminé avec le code de retour "
     <<  WEXITSTATUS(status) << endl;
```

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)`.

Exercice `fork + wait` : la course de haies (TP + Maison)

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"
```

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 :

```
srandom(time());
...
int r = a+ random() % (b-a+1);
```

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.

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()`.