

Castille Vincent
Lichnowski Nicolas
Mattio Jérémy

Groupe 3
2^{ème} année
2005/2006

Devoir de système:

Mini-Shell, Prof: Andréa Dragut:

Shell-a

Sommaire

1. Qu'est ce qu'un shell?

2. Méthodologie des classes

2.1. Classe Command

2.2. Classe Parseur, LecteurCaractere, LecteurSymbole
et Symbole

2.3 Classe Process

3. Les plus:

3.1. Redirection en entrée

3.2. Redirection vers la fin d'un fichier

3.3. Commande jobs

3.4. Scripts

4. Ce qu'il reste à faire

5. Annexes:

5.1. Codes sources

1.Qu'est ce qu'un shell?

Avant de commencer à expliquer quoi que se soit il nous est apparu nécessaire d'expliquer ce qu'est un shell et comment il fonctionne. Un shell c'est quoi? C'est un logiciel faisant partie des composants de base d'un système d'exploitation. Son rôle est de traiter des lignes de commande tapées au clavier. Ces commandes, une fois traitées et interprétées, auront pour effet de réaliser telle ou telle tâche d'administration, ou bien de lancer l'exécution d'un autre logiciel. De plus un shell contient plusieurs variables d'environnement comme par exemple HOME qui définit les chemins depuis la racine aux répertoires utilisateurs. Les commandes d'un shell peuvent être de plusieurs sortes et nous les étudierons plus bas.

Les commandes Simples:

Les commandes simples, ce sont toutes les commandes tapées avec ou sans argument, ou une seule commande sur la ligne de commande. Cependant on peut encore classer les commandes simples en deux catégories :

Les commandes externes :

Les commandes externes concernent toutes les commandes qui sont propres au système (elles sont répertoriées dans les répertoires /bin et /sbin).

Les commandes internes :

Les commandes internes, ce sont toutes les commandes qui sont propres au shell et qui peuvent être différentes d'un shell à l'autre.

Les commandes complexes :

Les commandes complexes, il s'agit en fait d'une suite de commandes regroupées en une seule. Une commande complexe est composée d'au moins une commande simple et d'une commande qui peut être simple ou complexe. Ces commandes sont regroupées de deux façons:

Les pipelines :

Ce sont des commandes simples enchaînées les unes à la suite des autres et séparées par des "|" (pipes), elles communiquent entre elles. Le résultat de l'une affecte l'entrée de l'autre.

Les suites :

Ce sont des commandes simples enchaînées les unes à la suite des autres et séparées par des ";" (points-virgules). La différence avec les pipelines est que, cette fois-ci, les commandes ne communiquent pas entre elles.

Maintenant nous allons entrer dans le vif du sujet et expliquer comment on a réfléchi pour concevoir ce mini shell.

2. Méthodologie des classes

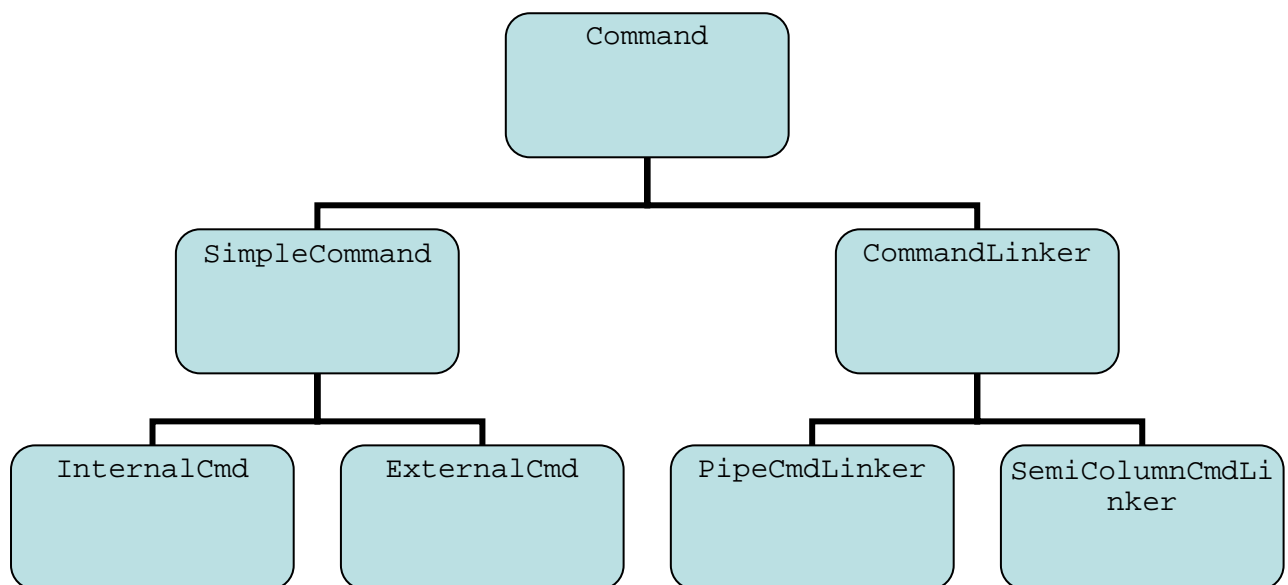
Au départ nous avons commencé par reprendre l'exo 3 du tp producteur/consommateur.

A force de modifier le code de cet exercice, nous sommes arrivés à un code illisible pour les autres (et pour nous aussi un peu) et dur à comprendre. Vincent a demandé de l'aide à un ami qui s'y connaît bien afin de jeter un coup d'œil à ce que nous avons fait. Il nous a conseillé de tout recommencer à zéro en nous recommandant de faire des classes, ce qui serait beaucoup plus simple.

Il a eu raison.

2.1. Classe Command

Pour commencer nous nous sommes occupés de l'exécution des Commandes sans se soucier du Parseur qui a été fait une fois que toutes les commandes s'exécutaient. Voici comment s'organisent nos classes de commandes :



La classe Command est une classe abstraite, effectivement elle possède une fonction `execute` virtuelle et abstraite qui est définie dans ses classes petites filles. Pour le moment, nous pouvons avoir l'impression de compliquer un peu le code mais par la suite ce sera plus simple pour instancier un objet de la classe Command ou d'une de ses petites filles.

Pour `InternalCmd` l'exécution de commandes ne nécessite pas de duplication de processus mais nécessite l'utilisation de fonctions C. Par exemple pour le `cd`, il faut utiliser `chdir` qui a un effet uniquement dans le processus courant. Donc si on duplique le processus pour faire

un `cd`, cela pose des problèmes car en revenant dans le père, on n'a pas changé de répertoire. On peut aussi imaginer modifier les variables d'environnement mais cela alourdirait le code... Sinon pour `InternalCmd` ce ne sont que des tests conditionnels pas très compliqués à faire. Petite précision avant de continuer, l'exécution en arrière plan ne signifie pas grand-chose pour les commandes internes, c'est pour cette raison qu'il n'y a pas de données membres `m_background` dans cette classe. Nous avons quand même remarqué sur le `tsh`, que si l'on essayait d'exécuter une commande interne en arrière plan cela créait un processus fils mais il nous apparut trop compliqué de l'implémenter dans ce devoir.

En revanche pour `ExternalCmd` on est obligé de dupliquer le processus puisque nous utilisons la fonction `execvp()` de laquelle on ne revient qu'en cas d'erreur comme par exemple si l'on tente d'exécuter un programme sur lequel on a pas les droits. Donc ici l'exécution en arrière plan à un sens significatif : c'est juste le fait d'attendre ou de ne pas attendre la fin d'un fils mais pas n'importe lequel. Effectivement nous avons eu des problèmes car nous attendions la fin de tous les fils dans une boucle `for` au moyen de `wait()` et du coup l'exécution en arrière plan ne marchait pas. (Ici on s'est inspiré de l'exo 3 du TP producteur/consommateur)

Exemple :

On peut imaginer exécuter une commande ou un programme en arrière plan et puis relancer une commande sans attendre la fin du fils précédent donc dans ce cas-la il faut bien savoir quel(s) fils attendre...

Nous avons donc décidé d'utiliser la fonction `waitpid()` pour pouvoir attendre la fin d'un fils bien précis et de récupérer le status ainsi nous pouvons tester si un fils s'est terminé normalement ou si il a été arrêté par un signal ou même encore savoir s'il est interrompu (`Suspended`).

En ce qui concerne les paramètres de la fonction `execute()` ils sont nécessaires uniquement pour l'exécution de commandes complexes (pipelines) et sont là juste pour la communication d'entre les processus de chaque commande. Dans `InternalCmd` nous n'avons pas besoin de nous en préoccuper car une commande interne avec pipeline n'a pas d'intérêt, par exemple on pourrait imaginer taper : "`cd .. | grep a`" mais cela n'a aucun sens...

Maintenant pour les `CommandLinker`, l'exécution n'est pas très compliquée. Pour `PipeCmdLinker` on s'est inspiré du TP. En créant un pipe et un file descriptor (pour la redirection du clavier) on voit l'intérêt des paramètres de la fonction `execute()`. `execute()` de `SemiColumnCmdLinker` se passe de commentaires puisqu'il suffit d'exécuter `lastcommand` puis `nextcommand`.

Une fois que toutes nos commandes marchaient, il a fallu faire le parseur. C'est là que l'ami de Vincent nous a été de bon conseil en nous disant de voir le sujet comme un tout et non comme une suite de questions.

2.2. Classe Parseur, LecteurCaractere, LecteurSymbole et Symbole

Le principe de base :

Notre parseur fait appel à la classe `LecteurSymbole` qui fait appel à la classe `Symbole` dans laquelle on a défini les tous les symboles à reconnaître comme par exemple un "|" ou un ";" mais aussi un ">>" pour la redirection vers la fin d'un fichier ce qui simplifie beaucoup les choses pour prendre en charge les scripts par la suite puisqu'il suffit de rajouter un symbole "if" et "fi" par exemple pour effectuer des tests conditionnels. Notre parseur fait aussi appel à un lecteur de caractère qui va lire comme son nom l'indique caractère par caractère.

La classe `LecteurCaractere`, qui lit des caractères, se construit à partir d'une chaîne de caractères ou d'un fichier, et permet de lire les caractères d'une chaîne un par un. Cette classe possède deux classes filles `LecteurCaractereFile` et `LecteurCaractereString`. Ses méthodes sont :

- Un constructeur et un destructeur virtuel de `LecteurCaractere`.
- Une méthode permettant de récupérer le caractère courant.
- Une méthode récupérant le numéro de ligne courante (pratique dans les scripts).
- Une méthode récupérant le numéro de la colonne courante, elle permet de connaître à quelle endroit dans une ligne de commande on a fait une erreur (en cours d'implémentation).
- Une méthode permettant de passer au caractère suivant. Elle est virtuelle et abstraite.

Les deux classes filles `LecteurCaractereFile` et `LecteurCaractereString` permettent respectivement de lire dans un fichier ou dans une chaîne de caractères.

Une classe qui lit des symboles, qui se construit de la même manière que la précédente, mais qui construit un lecteur de caractères et s'en sert pour créer des instances d'une autre classe `Symbole`. Ce dernier objet représente un élément de la ligne de commande, il possède un code qui définit son type :

Déclaration : `enum { INDEFINI=0, FIN, CHAINE, PIPE, BACKGROUNDED, POINT_VIRGULE, INFERIEUR, SUPERIEUR, SUPERIEUR_DBL };`

En résumé, un code par symbole élémentaire que l'on doit reconnaître. `CHAINE` représente toutes les chaînes que l'on peut reconnaître, qu'elles soient des noms de commandes ou des arguments de ces commandes.

- Une méthode pour récupérer une chaîne de caractères,
- Une méthode pour récupérer le code d'un symbole,
- Surcharge des opérateurs `==` et `!=` pour pouvoir effectuer les tests nécessaires à la reconnaissance des symboles.
- Surcharge de l'opérateur `<<` qui affiche un symbole en fonction de son code.

Donc ce que le lecteur de symbole doit faire, c'est utiliser le lecteur de caractère qu'il a construit pour reconnaître des symboles. Ses méthodes doivent être :

- Un constructeur `LecteurSymbole` qui prend en paramètre une `NTCTS` et un booléen. La chaîne de caractères représente soit une commande soit le nom d'un fichier (fonction du booléen),
- Une méthode retournant le symbole courant,
- Une méthode permettant de retourner le caractère courant,
- Une méthode permettant de retourner la ligne courante,
- Une méthode permettant de retourner la colonne courante,
- Une méthode permettant de "pointer" sur le symbole suivant. Cette méthode utilise une méthode `sauterSeparateur` qui saute les séparateurs (espace, tabulation...).
- Une méthode de test du symbole courant (test de son code, donc de son type...) qui retourne vrai si le code du symbole courant est bien égal à celui donné,
- Une méthode qui lit grâce au lecteur de caractère la chaîne du prochain symbole et la renvoie en résultat.

Une classe `Parseur` qui s'occupe du découpage d'une ligne de commandes en commandes simples puis après découpe une commande simple en fonctions des différents caractères d'espacements (exemple : découpe "ls -l" en "ls", "-l"). Cette classe possède une donnée membre (lecteur de symboles) et plusieurs méthodes :

- Un constructeur qui prend en paramètre une `NTCTS` et un booléen. La chaîne de caractères représente soit une commande soit le nom d'un fichier (fonction du booléen),
- Une méthode `parse` qui est analogue à la fonction `AnalyserCommand` du tp et qui construit une commande complexe.
- Une méthode qui permet de construire une commande simple.

2.3 Classe Process

Cette classe permet la gestion des processus et tout ce qui s'y rattache, c'est-à-dire, les fichiers, l'état (stoppé ou non). Elle possède différentes méthodes dont quelques accesseurs et modifieurs. Les données membres sont : un pid, un fichier d'entrée, un fichier de sortie, l'état (stoppé ou non), un booléen d'arrière-plan ou non et le plus important un `vector static` contenant tous les processus lancés par le shell.

Un dernier mot sur le jobs control et les signaux. Il faut impérativement dérouter le signal `SIGCHLD` parce que sinon on risque de ne pas s'apercevoir de la mort d'un fils en arrière plan et du coup on ne pourra pas quitter. Nous avons rajouter une condition avant de quitter le shell (on vérifie qu'il n'y a plus de processus enregistré dans le `vector` de `Process`).

3. Les plus:

3.1. Redirection en entrée

Ayant implémenté comme le sujet le précise la redirection dans un fichier, nous avons pensé utile d'implémenter la redirection en entrée qui ne change pas grand-chose au code, on redirige juste l'entrée standard 0 (clavier) vers un fichier.

3.2. Redirection vers la fin d'un fichier

La redirection vers la fin d'un fichier nous a paru utile dans certains cas, c'est pour ça que nous l'avons implémenté. C'est une simple option dans la fonction système `open()`.

3.3. Commande jobs

Cette commande permet d'afficher tous les processus en cours d'exécution.

3.4. Scripts

Pour les scripts tout ne marche pas, on prend en charge un fichier de script comme ligne de commande mais il nous reste une grosse tache à accomplir qui est de définir le langage. Par exemple, définir des symboles (`if`, `for`, `while`,...) dans la classe `Symbole`. Il faudra aussi préciser que tel ou tel symbole doit être suivi d'un autre (exemple : on peut imaginer qu'un `if` doit être suivi d'une parenthèse ouvrante, puis d'une chaîne de caractère et d'une parenthèse fermante). Ce qui n'est pas évidemment à faire.

4. Ce qui reste à faire

Dans ce devoir, toutes les exceptions ne sont pas traitées ou catchées par manque de temps, quelques unes seulement ont été traitée (exemple: à plusieurs endroit dans le code on effectue des tests et l'on affiche un message d'erreur au lieu de lever une exception). Nous voulions aussi créer un historique des commandes tapées mais nous n'avons pas eu le temps. De même que l'affichage du répertoire courant dans le prompt.

Actuellement il y a quelques fonctionnalités en cours:

- Les signaux provenant du clavier. Notre shell ne réagit pas à ces signaux mais le fils en cours d'exécution ne le reçoit pas car on n'a pas eu le temps de l'implémenter.
- Les scripts. Pour pouvoir faire fonctionner les scripts, il nous reste pas mal de travail. Il faut que l'on crée de nouvelles Commandes interne (ex: "if", "fi",...). De plus il faut indiquer que tel mot clé doit être suivi d'un `Symbole` ou d'un autre mot clé bien précis. Il faut en gros construire notre propre langage. Il nous semblerait judicieux d'implémenter cette fonctionnalité sur le principe des arbres.

5. Annexes:

5.1. Codes sources

Pour les codes sources nous fournissons que les fichiers que l'on a créés ainsi ceux que l'on a créés en TP ne seront pas présents dans ce listing.

Include

```
/*
 *
 * @File: Command.h
 *
 *
 * @Date: 23/12/05
 *
 *
 * @Authors: Castille, Mattio, Lichnowski
 *
 *
 */

#ifndef __COMMAND_H__
#define __COMMAND_H__

//Représente une commande (une ou plusieurs commandes simples...).

class Command
{
public:
    Command();

    virtual void print(unsigned char indent = 0);

    //Exécute la commande entière.

    virtual void execute(pid_t & pgrp, int fromPipe = -1,
                        int intoPipe = -1) = 0;
}; //Command

class SimpleCommand : public Command
{
protected:
    bool m_background;
    unsigned int m_argc;
    char** m_argv;
    char *m_inRedirection, *m_outRedirection;
    bool m_append;

public:
    SimpleCommand(unsigned int argc, char* argv[], bool
background);

    void setBackground();
    void setInRedirection(char *fileName);
};
```

```

        void setOutRedirection(char *fileName, bool append = false);

        void print(unsigned char indent = 0);

        static SimpleCommand* buildCommand(unsigned int argc,
                                           char* argv[]);
}; //SimpleCommand

//tout ce qui ne duplique pas le processus i.e. cd, fg, sendsig :
//peut tout de même être considéré comme "mettable" en arrière plan,
//même si ça n'a aucun effet.

class ExternalCmd : public SimpleCommand
{
    public:
        ExternalCmd(unsigned int argc, char** argv,
                    bool background = false);
        void execute(pid_t & pgrp, int fromPipe = -1,
                    int intoPipe = -1);
}; //ExternalCmd

class InternalCmd : public SimpleCommand
{
    public:
        InternalCmd(unsigned int argc, char* argv[],
                    bool background = false);
        void execute(pid_t & pgrp, int fromPipe = -1,
                    int intoPipe = -1);
}; //InternalCmd

//Commande complexe avec un | ou un ;

class CommandLinker : public Command
{
    protected:
        Command *m_lastCommand;
        SimpleCommand *m_nextCommand;

        virtual char* getSymbol() = 0;

    public:
        CommandLinker(Command *lastCommand,
                      SimpleCommand *nextCommand);

        void print(unsigned char indent = 0);
}; //CommandLinker

class PipeCmdLinker : public CommandLinker

```

```

{
    public:
        PipeCmdLinker(Command *lastCommand,
                      SimpleCommand *nextCommand);

        void execute(pid_t & pgrp, int fromPipe = -1,
                    int intoPipe = -1);

        char* getSymbol();
}; //PipeCmdLinker

class SemiColumnCmdLinker : public CommandLinker
{
    public:
        SemiColumnCmdLinker(Command *lastCommand,
                            SimpleCommand *nextCommand);

        void execute(pid_t & pgrp, int fromPipe = -1,
                    int intoPipe = -1);

        char* getSymbol();
}; //SemiColumnCmdLinker

#include "Command.hxx"
#endif /* __COMMAND_H__ */

```

```

/*
 *
 *
 * @File: Command.hxx
 *
 *
 * @Date: 23/12/05
 *
 *
 * @Authors: Castille, Mattio, Lichnowski
 *
 *
 */

```

```

#ifndef __COMMAND_HXX__
#define __COMMAND_HXX__
#include "Command.h"

```

```

inline void SimpleCommand::setBackground()
{
    m_background = true;
} //setBackground()

```

```

inline void SimpleCommand::setInRedirection(char *fileName)
{
    m_inRedirection = fileName;
} //setInRedirection()

```

```

inline void SimpleCommand::setOutRedirection(char *fileName,

```

```

        bool append/* = false*/)
{
    m_outRedirection = fileName; m_append = append;
} //setOutRedirection()

inline char* PipeCmdLinker::getSymbol() { return "|"; }
inline char* SemiColumnCmdLinker::getSymbol() { return ";"; }

#endif /* __COMMAND_H__ */

/*
 *
 *
 * @File: LecteurCaractere.h
 *
 *
 * @Date: 23/12/05
 *
 *
 * @Authors: Castille, Mattio, Lichnowski
 *
 *
 */

#ifndef __LECTEUR_CARACTERE_H__
#define __LECTEUR_CARACTERE_H__

#include <fstream>
using namespace std;

class LecteurCaractere
{
public :
    LecteurCaractere(); // construit le lecteur
    virtual ~LecteurCaractere() {}
    char getCarCour();
    unsigned int getLigne();
    unsigned int getColonne();
    virtual void suivant() = 0; // passe au caractere suivant, s'il
existe

protected :
    unsigned int colonne; // numéro de colonne du caractère courant
    unsigned int ligne; // numéro de ligne du caractère courant

    char carCour; // le caractère courant
}; //LecteurCaractere

class LecteurCaractereFile : public LecteurCaractere
{
public :
    LecteurCaractereFile(const char *str);
    virtual ~LecteurCaractereFile() {}

```

```

        void suivant();

    private :
        ifstream f;    // le fichier texte que l'on parcourt
}; //LecteurCaractereFile

////////////////////////////////////
class LecteurCaractereString : public LecteurCaractere
{

    public :
        LecteurCaractereString(const char *str);
        virtual ~LecteurCaractereString() {}
        void suivant();

    private :
        unsigned int currentIndex;
        const char *str;
}; //LecteurCaractereString

#include "LecteurCaractere.hxx"
#endif /* __LECTEUR_CARACTERE_H__ */

/**
 * @File      : LecteurCaractere.hxx
 *
 * @Authors   : Castille, Lichnowski, Mattio
 *
 * @Date: 23/12/05
 *
 */

#ifndef __LECTERECARACTERE_HXX__
#define __LECTERECARACTERE_HXX__

#include "LecteurCaractere.h"

inline char LecteurCaractere::getCarCour() { return carCour; }
inline unsigned int LecteurCaractere::getLigne() { return ligne; }
inline unsigned int LecteurCaractere::getColonne() { return colonne; }

#endif /* __LECTERECARACTERE_H__ */

```

```

/**
 * @File      : LecteurSymbole.h
 *
 * @Authors   : Castille, Lichnowski, Mattio
 *
 * @Date: 23/12/05
 *
 **/

#ifndef __LECTEURSYMBOLE_H__
#define __LECTEURSYMBOLE_H__

#include "LecteurCaractere.h"
#include "Symbole.h"

#include <fstream>
using namespace std;

class LecteurSymbole
{
public:
    // résultat : symCour = premier symbole de la chaine str
    LecteurSymbole(const char str[], bool file);

    char getCar();

    // renvoie le symbole courant
    Symbole getSymCour();

    // renvoie la ligne du symbole courant
    unsigned int getLigne();

    // renvoie la colonne du symbole courant
    unsigned int getColonne();

    void suivant(); // passe au symbole suivant du fichier

    // si symCour!=code, on arrete le programme, sinon rien
    void testerSymCour (unsigned short code);

    // si symCour==code, on passe au symbole suivant, sinon on
arrete
    void sauterSymCour (unsigned short code);

    char* getChaine();

private:
    LecteurCaractere *lc; // le lecteur de caracteres
    Symbole symCour; // le symbole courant du lecteur de symboles

    // coordonnees, dans le fichier, du symbole courant
    unsigned int ligne, colonne;

    // saute dans lc une suite de separateurs consecutifs
    void sauterSeparateurs ();

    // lit, grace à lc, la chaine du prochain symbole et
    //la renvoie en resultat
    char* motSuivant ();
}; //LecteurSymbole

```

```

#include "LecteurSymbole.hxx"
#endif /* __LECTEURSYMBOLE_H__ */

/**
 * @File      : LecteurSymbole.hxx
 *
 * @Authors   : V. Castille
 *
 * @Date: 23/12/05
 *
 */

#ifndef __LECTEURSYMBOLE_HXX__
#define __LECTEURSYMBOLE_HXX__

#include "LecteurSymbole.h"

inline char LecteurSymbole::getCar() {return lc->getCarCour(); }
inline Symbole LecteurSymbole::getSymCour() {return symCour; }
inline unsigned int LecteurSymbole::getLigne() {return ligne; }
inline unsigned int LecteurSymbole::getColonne() {return colonne; }

#endif /* __LECTEURSYMBOLE_H__ */

/*
 *
 * @File: Parseur.h
 *
 *
 * @Date: 23/12/05
 *
 *
 * @Authors: Castille, Mattio, Lichnowski
 *
 *
 */
#ifndef __PARSEUR_H__
#define __PARSEUR_H__

#include "LecteurSymbole.h"
#include "Command.h"

class Parseur
{
public:
    Parseur(const char *str, bool file);
    Command *parse();
};

```

```

        private:
            SimpleCommand* getSimpleCommand();
            LecteurSymbole ls;
}; //Parseur

#endif /* __PARSEUR_H__ */

/**
 *
 * @File: Process.h
 *
 *
 * @Date: 23/12/05
 *
 *
 * @Authors: Castille, Mattio, Lichnowski
 *
 *
 */

#ifndef __PROCESS_H__
#define __PROCESS_H__

#include "nsSysteme.h"
#include <vector>
#include <iostream>

class Process
{
    pid_t m_pid, m_pgrp;
    char **m_argv;
    bool m_completed;
    bool m_stopped;
    int m_status;
    bool m_background;

public:
    static std::vector<Process*> TableProcess;
    Process (char **argv, pid_t pgrp, bool bg = false);
    void launch(int infd, int outfd, int errfd);
    void setForeground(bool continuing);
    void setBackground(bool continuing);
    void setCompleted();
    pid_t getPID();
    bool isBackground();
    void setPID(pid_t pid);

    bool isStopped();
    bool isCompleted();

    static Process* getProcessFromPID(pid_t pid);
    static int getProcessIndexFromPID(pid_t pid);

```

```

        static Process* getProcessAt(unsigned int pos);
        static Process* getLastProcess();
        static void deleteLastProcess();
        static void deleteProcessAt(unsigned int pos);

        friend std::ostream & operator << (std::ostream &, Process);

}; //Process
#include "Process.hxx"
#endif /* __PROCESS_H__ */

/**
 *
 * @File: Process.hxx
 *
 *
 * @Date: 23/12/05
 *
 *
 * @Authors: Castille, Mattio, Lichnowski
 *
 *
 */

#ifndef __PROCESS_HXX__
#define __PROCESS_HXX__

#include "Process.h"

inline void Process::setPID(pid_t pid) {this->m_pid = pid;}

inline bool Process::isStopped() {return this->m_stopped;}
inline bool Process::isCompleted() {return this->m_completed;}

#endif /* __PROCESS_HXX__ */

/**
 *
 * @File: Symbole.h
 *
 *
 * @Date: 23/12/05
 *
 *
 * @Authors: Castille, Mattio, Lichnowski
 *
 *
 */

```

```

#ifndef __SYMBOLE_H__
#define __SYMBOLE_H__

#include <iostream>
using namespace std;

//class Symbole
//brief La classe représentant les symboles.

////////////////////////////////////
namespace nsSymboles_Code {
    enum {INDEFINI = 0,
          FIN,
          CHAINE,
          PIPE,
          BACKGROUNDED,
          POINT_VIRGULE,
          INFERIEUR,
          SUPERIEUR,
          SUPERIEUR_DBL
    }; //nsSymboles_Code
}
using namespace nsSymboles_Code;
////////////////////////////////////

class Symbole
{
    //////////////////////////////////////
    //public:

    //fn Symbole
    //param ch la chaine représentée par le symbole à construire.

        Symbole (char ch[]=""); // construit le symbole représenté par ch

    //fn Symbole
    //param code le code représentant le symbole à construire.

        Symbole(unsigned short code=INDEFINI); //construit le symbole de code
        indique

        inline char* getChaine ();
        inline unsigned short getCode();
        inline int operator == (unsigned short code);
        inline int operator != (unsigned short code);

        friend ostream & operator << (ostream & cout, Symbole symb);

        // affiche symb sur cout

    //////////////////////////////////////
    //protected:
        static const unsigned int NB_SYMBILES = 9;
        // nombre de symboles dans le langage

```

```

static const unsigned int TAILLE_MAX_CHAINE_CODE = 3;
// taille max. de la chaine d'un symbole :
//à agrandir si symboles spéciaux de plusieurs caractères...
//en comptant le '\0'

static const unsigned int TAILLE_MAX_CHAINE = 512;
// taille max. d'une chaine de caractères

static const char motCle[NB_SYMBOLLES][TAILLE_MAX_CHAINE_CODE];
// motCle[i] = chaine du symbole de code i

static const char codeEnClair[NB_SYMBOLLES][15];
// codeEnClair[i] = chaine utilisée dans une trace pour
//le symbole de code i

////////////////////////////////////
/////

private:

    char  chaine[TAILLE_MAX_CHAINE]; // chaine du symbole

    unsigned short code;           // code du symbole
}; //CSymbole

#include "Symbole.hxx"
#endif /* __SYMBOLE_HXX__ */

/**
 *
 * @File: Symbole.h
 *
 *
 * @Date: 23/12/05
 *
 *
 * @Authors: Castille, Mattio, Lichnowski
 *
 *
 */

#ifndef __SYMBOLE_HXX__
#define __SYMBOLE_HXX__

#include "Symbole.h"

inline char* Symbole::getChaine ()
{
    return this->chaine;
} //getChaine

inline unsigned short Symbole::getCode()

```

```

{
    return this->code;
} //getCode

inline int Symbole::operator == (unsigned short code)
{
    return this->code==code;
} //operator ==

inline int Symbole::operator != (unsigned short code)
{
    return this->code!=code;
} //operator !=

#endif /* __SYMBOLE_H__ */

    #/**
**
** @File : INCLUDE_H
**
** @Authors : D. Mathieu, M. Laporte
**
** @Date : 30/07/2001
**
** @Version : V1.0
** @Author : A. B. Dragut
**
** @Synopsis : Dépendances des différents fichiers inclus
**
** @Modified :
**     @Authors : D. Mathieu, M. Laporte
**     @Date : 24/10/2001
**     @Synopsis : ajout de la macro PPAL_H pour compatibilité avec
**                 gcc 3.2
***/
INCLUDE = ../include
UTIL     = ../util
#
#
PPAL_H      = $(INCLUDE)/ppal.h
CEDITABLE_H = $(INCLUDE)/CEditable.h          $(CEDITABLE_HXX)
#
CSTCODERR_H = $(INCLUDE)/CstCodErr.h
#
CEXCEPTION_HXX = $(INCLUDE)/CException.hxx    $(CSTCODERR_H)
CEXCEPTION_H   = $(INCLUDE)/CException.h      $(CEDITABLE_H)          \
                                                    $(CSTCODERR_H)          \
                                                    $(CEXCEPTION_HXX)
#
CEXCFCTSYST_HXX = $(INCLUDE)/CExcFctSyst.hxx  $(CEXCEPTION_H)
CEXCFCTSYST_H   = $(INCLUDE)/CExcFctSyst.h    $(CEXCEPTION_H)          \
                                                    $(CEXCFCTSYST_HXX)

NSSYSTEME_HXX   = $(INCLUDE)/nsSysteme.hxx    $(CEXCFCTSYST_H)

```

```

NSSYSTEME_H      = $(INCLUDE)/nsSysteme.h          $(NSSYSTEME_HXX)      \
                                                         $(CEXCEPTION_H)      \
                                                         $(CEXCFCTSYST_H)
#
NSUTIL_H         = $(INCLUDE)/nsUtil.h            $(CEXCEPTION_H)
#
PROCESS_HXX     = $(INCLUDE)/Process.hxx
PROCESS_H       = $(INCLUDE)/Process.h            $(NSSYSTEME_HXX) $(PROCESS_HXX)
#
COMMAND_HXX     = $(INCLUDE)/Command.hxx
COMMAND_H       = $(INCLUDE)/Command.h           $(COMMAND_HXX)
#
SYMBOLE_HXX    = $(INCLUDE)/Symbole.hxx
SYMBOLE_H      = $(INCLUDE)/Symbole.h           $(SYMBOLE_HXX)
#
LECTEUR_CARACTERE_HXX = $(INCLUDE)/LecteurCaractere.hxx
LECTEUR_CARACTERE_H = $(INCLUDE)/LecteurCaractere.h
                  $(LECTEUR_CARACTERE_HXX)
#
LECTEUR_SYMBOLE_HXX = $(INCLUDE)/LecteurSymbole.hxx
LECTEUR_SYMBOLE_H = $(INCLUDE)/LecteurSymbole.h  $(SYMBOLE_H)          \
                  $(LECTEUR_CARACTERE_H)
                  $(LECTEUR_CARACTERE_HXX)
#
PARSEUR_H      = $(INCLUDE)/Parseur.h            $(LECTEUR_SYMBOLE_H)  \
                                                         $(COMMAND_H)

```

Util

```

/*
 *
 * @File: Command.cxx
 *
 *
 * @Date: 23/12/05
 *
 *
 * @Authors: Castille, Mattio, Lichnowski
 *
 *
 */

```

```

#include <string>
#include <iostream>
#include <csignal>
#include "Command.h"
#include "nsSysteme.h"
#include "Process.h"

```

```

using namespace nsSysteme;
using namespace std;

```

```

Command::Command() { }

```

```

//pour le debogage
void Command::print(unsigned char indent)
{
    for (unsigned int i = 0; i < indent; i++)
        cout << "  ";
}

SimpleCommand::SimpleCommand(unsigned int argc, char* argv[], bool
background)
    : Command(), m_background(background), m_argc (argc), m_argv (argv)
{
    m_inRedirection = NULL;
    m_outRedirection = NULL;
}

//pour le debogage
void SimpleCommand::print(unsigned char indent)
{
    Command::print(indent);
    for (unsigned int i = 0; i < this->m_argc; i++)
        cout << this->m_argv[i] << ' ';
    cout << endl;
}

//Cette fonction doit détecter les commandes internes...
SimpleCommand* SimpleCommand::buildCommand(unsigned int argc, char* argv[])
{
    if (argc < 1)
        return NULL;
    unsigned int length = strlen(argv[0]);
    switch (length)
    {
        case 2:
            if (strcmp(argv[0], "cd") == 0)
                return new InternalCmd(argc, argv);
            if (strcmp(argv[0], "fg") == 0)
                return new InternalCmd(argc, argv);
            if (strcmp(argv[0], "bg") == 0)
                return new InternalCmd(argc, argv);
            break;

        case 4:
            if (strcmp(argv[0], "exit") == 0)
                return new InternalCmd(argc, argv);

            else if (strcmp(argv[0], "jobs") == 0)
                return new InternalCmd(argc, argv);
            break;

        case 5:
            if (strcmp(argv[0], "clear") == 0)
                return new InternalCmd (argc, argv);
            break;

        case 7:
            if (strcmp(argv[0], "sendsig") == 0)
                return new InternalCmd(argc, argv);
            break;
    }

}

```

```
    return new ExternalCmd(argc, argv);
} //buildCommand
```

```
InternalCmd::InternalCmd(unsigned int argc, char* argv[], bool background)
    : SimpleCommand(argc, argv, background) {}
```

```
void InternalCmd::execute(pid_t & pgrp, int fromPipe /*= -1*/,
                          int intoPipe /*= -1*/)
{
    int Size = strlen(m_argv[0]);

    switch (Size)
    {
    case 2:
        if (strcmp(m_argv[0], "cd")==0)
            if (m_argc == 2)
                chdir (m_argv[1]);
            else
                chdir (getenv("HOME"));
        else if (strcmp (m_argv[0], "fg") == 0)
        {
            Process *p;
            if (m_argc == 2)
                if (m_argv[1][0] == '%')
                    p = Process::getProcessAt(atoi(&
                                                    (m_argv[1][1]));)
                else
                    p = Process::getProcessFromPID(pid_t(
                                                    atoi(m_argv[1])));

            else
                p = Process::getLastProcess();

            p->setForeground(true);

        } //if
        else if (strcmp (m_argv[0], "bg") == 0)
        {

            Process *p;
            if (m_argc == 2)
                if (m_argv[1][0] == '%')
                    p = Process::getProcessAt(atoi(&
                                                    (m_argv[1][1]));)
                else
                    p = Process::getProcessFromPID(pid_t(
                                                    atoi(m_argv[1])));

            else
                p = Process::getLastProcess();

            p->setBackground(true);

        } //if
        break;

    case 4:
        if (strcmp(m_argv[0], "jobs") == 0)
        {
```

```

        for (unsigned i = 0; i < Process::TableProcess
                .size(); ++i)
        {
            cout << '[' << i << "]" << " "
                << (Process::TableProcess.at(i))
                << "->getPID() << '\n';
        }//for
    }//if
    else if (strcmp(m_argv[0], "exit") == 0)
    {
        if (Process::TableProcess.size() >= 1)
            cerr << "Impossible de quitter"
                << "(creation de demon!!!)\n";
        else
            exit (0);
    }//if
    break;

    case 5:
        if (strcmp(m_argv[0], "clear") == 0)
            cout << "\033[2J\033[0;0f"; //efface l'écran

    case 7:
        if (strcmp(m_argv[0], "sendsig") == 0)
            if (m_argc == 3)
            {
                if (m_argv[2][0] == '%')
                    kill (Process::getProcessAt(atoi(
                        &m_argv[2][1]))->getPID(),
                        atoi(&m_argv[1][1]));
                else
                    kill (pid_t(atoi(m_argv[2])),
                        atoi(&m_argv[1][1]));
            }
            //if

            else cerr << "Usage: sendsig -<No_signal>"
                << " [%job] [pid]\n";
        break;

    }//switch

} //InternalCmd::execute

ExternalCmd::ExternalCmd(unsigned int argc, char** argv, bool background) :
    SimpleCommand(argc, argv, background) {}

void ExternalCmd::execute(pid_t & pgrp, int fromPipe /*= -1*/,
                        int intoPipe /*= -1*/)
{
    int fdficIn = -1, fdficOut = -1;

    if (this->m_inRedirection != NULL)
    {
        // redirection en entrée
        fdficIn = Open(this->m_inRedirection, O_RDONLY, 0700);
    }//if

    if (this->m_outRedirection != NULL)

```

```

{
    // redirection en sortie
    if (this->m_append)
        fdficOut = Open(this->m_outRedirection,
                        O_WRONLY | O_CREAT | O_APPEND, 0700);
    else
        fdficOut = Open(this->m_outRedirection,
                        O_WRONLY | O_CREAT | O_TRUNC, 0700);
} //if

if (fromPipe != -1)
    fdficIn = fromPipe;
if (intoPipe != -1)
    fdficOut = intoPipe;

pid_t pid;
Process* p = new Process(this->m_argv, pgrp, this->m_background);

sigset_t fill, old;

sigfillset(&fill);
Sigprocmask(SIG_BLOCK, &fill, &old);

switch (pid = Fork())
{
    case -1:
        Sigprocmask(SIG_SETMASK, &old, NULL);
        // error !
        return;
        break;

    case 0: // child...
        Sigprocmask(SIG_UNBLOCK, &fill, NULL);
        p->launch(fdficIn, fdficOut, /*STDERR_FILENO*/-1/*,
                !this->m_background*/);
        break;

    default: // parent...
        p->setPID(pid);
        if (pgrp == 0)
            pgrp = pid;
        setpgid(pid, pgrp);

        if (this->m_background)
            cout << "[" << Process::getProcessIndexFromPID(pid)
                << "]" << pid << endl;
} //switch

if (this->m_background)
    p->setBackground(false);
else
    p->setForeground(false);

Sigprocmask(SIG_SETMASK, &old, NULL);

} //ExternalCmd::execute

CommandLinker::CommandLinker(Command *lastCommand, SimpleCommand
*nextCommand)

```

```

        : Command(), m_lastCommand (lastCommand), m_nextCommand (nextCommand)
    {}

//pour le debogage
void CommandLinker::print(unsigned char indent)
{
    Command::print(indent);
    cout << "command linker : " << this->getSymbol() << endl;
    this->m_lastCommand->print(indent + 1);
    if (this->m_nextCommand != NULL)
        this->m_nextCommand->print(indent + 1);
}

PipeCmdLinker::PipeCmdLinker(Command *lastCommand, SimpleCommand
*nextCommand)
    : CommandLinker(lastCommand, nextCommand) {}

void PipeCmdLinker::execute(pid_t & pgrp, int fromPipe /*= -1*/,
                                                                    int
intoPipe /*= -1*/)
{
    int fdClavier = Dup (0);
    int pfd[2];
    Pipe(pfd);

    this->m_lastCommand->execute(pgrp, fromPipe, pfd[1]);
    Close(pfd[1]);
    this->m_nextCommand->execute(pgrp, pfd[0], intoPipe);
    Close(pfd[0]);

    Dup2 (fdClavier, 0);
}

SemiColumnCmdLinker::SemiColumnCmdLinker(Command *lastCommand,
SimpleCommand *nextCommand)
    : CommandLinker(lastCommand, nextCommand) {}

void SemiColumnCmdLinker::execute(pid_t & pgrp, int fromPipe /*= -1*/,
                                                                    int intoPipe /*= -1*/)
{
    this->m_lastCommand->execute(pgrp, fromPipe);
    if (this->m_nextCommand != NULL)
        this->m_nextCommand->execute(pgrp, -1, intoPipe);
}

```



```

void LecteurSymbole::suivant()
{
    sauterSeparateurs();
    ligne = lc->getLigne();
    colonne = lc->getColonne();
    symCour = Symbole(motSuivant());
} //suivant()

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void LecteurSymbole::testerSymCour(unsigned short code)
{
    if (symCour != code) exit(1);
} //testerSymCour

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void LecteurSymbole::sauterSymCour(unsigned short code)
{
    if (symCour != code) exit(1);
    lc->suivant();
} //sauterSymCour

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void LecteurSymbole::sauterSeparateurs()
{
    while (lc->getCarCour() == ' ' || lc->getCarCour() == '\t' ||
           lc->getCarCour() == '\n' || lc->getCarCour() == '\r')
        lc->suivant();
    if (lc->getCarCour() == '#')
    {
        unsigned int ligne = lc->getLigne();
        while (lc->getCarCour() != EOF && lc->getLigne() == ligne)lc-
>suivant();
        sauterSeparateurs();
    } //if
} //sauterSeparateurs()

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
char * LecteurSymbole::motSuivant()
{
    const short TAILLE_SPECIAL_MAX = 3;
    const short TAILLE_MAX=512;
    static char chaine[TAILLE_MAX]; //doit etre static pour etre renvoyee
                                     //en resultat
    int nbCar = 0;

    sauterSeparateurs();

    if (isalpha(lc->getCarCour()) ||
        isdigit(lc->getCarCour()) ||
        lc->getCarCour() == '.' ||
        lc->getCarCour() == '/' ||
        lc->getCarCour() == '-' ||
        lc->getCarCour() == '_' ||
        lc->getCarCour() == '~')

```

```

do
{
    chaine[nbCar++]=lc->getCarCour();
    lc->suivant();
} //do
while (nbCar < TAILLE_MAX - 1 && (isalpha(lc->getCarCour()) ||
    isdigit(lc->getCarCour()) ||
    lc->getCarCour() == '.' ||
    lc->getCarCour() == '/' ||
    lc->getCarCour() == '-' ||
    lc->getCarCour() == '_' ||
    lc->getCarCour() == '=' ||
    lc->getCarCour() == '*' ||
    lc->getCarCour() == '~'));

else if (lc->getCarCour()=='')
{
    lc->suivant();
    while (lc->getCarCour() != EOF && lc->getCarCour() != '')
    {
        chaine[nbCar++] = lc->getCarCour();
        lc->suivant();
    } //while

    if (lc->getCarCour() == '')
        lc->suivant();

} // if
else if (lc->getCarCour() != EOF)
{
    if (lc->getCarCour() == ';')
    {
        chaine[nbCar++]=lc->getCarCour();
        lc->suivant();
    } //if
    else
    {
        do
        {
            chaine[nbCar++]=lc->getCarCour();
            lc->suivant();
        } //do
        while (nbCar < TAILLE_SPECIAL_MAX - 1 &&
            (lc->getCarCour() == '|' ||
            lc->getCarCour() == '&' ||
            lc->getCarCour() == '<' ||
            lc->getCarCour() == '>'));
    } //else
} //if

chaine[nbCar]='\0';
return chaine;
} //LecteurSymbole.cxx

```

```

/*
 *
 * @File: Parseur.cxx
 *
 *
 * @Date: 23/12/05
 *
 *
 * @Authors: Castille, Mattio, Lichnowski
 *
 *
 */

#include "Parseur.h"
#include "Symbole.h"
#include <string.h>

using namespace nsSymboles_Code;

#define ARGS_ALLOCATION_STEP 10

SimpleCommand* Parseur::getSimpleCommand()
{
    unsigned int argc = 0, currentAllocated = 0;
    char **argv = NULL;

    while (ls.getSymCour().getCode() == CHAINE)
    {
        if (argc == currentAllocated)
        {
            currentAllocated += ARGS_ALLOCATION_STEP;
            argv = (char**)realloc(argv, sizeof(char*) *
                                   currentAllocated);
        }
        //if
        char * str = (char*)malloc(sizeof(char) *
                                   strlen(ls.getSymCour().getChaine()));

        strcpy(str, ls.getSymCour().getChaine());
        argv[argc++] = str;
        ls.suivant();
    } //while

    if (argc == 0)
        return NULL;

    argv[argc + 1] = NULL;

    SimpleCommand* cmd = SimpleCommand::buildCommand(argc, argv);

    while (ls.getSymCour().getCode() == SUPERIEUR ||
           ls.getSymCour().getCode() == SUPERIEUR_DBL ||
           ls.getSymCour().getCode() == INFERIEUR)
    {
        switch (ls.getSymCour().getCode())
        {
            case SUPERIEUR:
                ls.suivant();
        }
    }
}

```

```

        if (ls.getSymCour().getCode() == CHAINE)
        {
            char * str = (char*)malloc(sizeof(char) *
                strlen(ls.getSymCour().getChaine()));

            strcpy(str, ls.getSymCour().getChaine());
            cmd->setOutRedirection(str);
            ls.suivant();
        }//if
        else
            cerr << "erreur de syntaxe caractere "
                << ls.getColonne() << endl;
        break;

    case SUPERIEUR_DBL:
        ls.suivant();
        if (ls.getSymCour().getCode() == CHAINE)
        {
            char * str = (char*)malloc(sizeof(char) *
                strlen(ls.getSymCour().getChaine()));

            strcpy(str, ls.getSymCour().getChaine());
            cmd->setOutRedirection(str, true);
            ls.suivant();
        }//if
        else
            cerr << "erreur de syntaxe caractere "
                << ls.getColonne() << endl;

        break;

    case INFERIEUR:
        ls.suivant();
        if (ls.getSymCour().getCode() == CHAINE)
        {
            char * str = (char*)malloc(sizeof(char) *
                strlen(ls.getSymCour().getChaine()));

            strcpy(str, ls.getSymCour().getChaine());
            cmd->setInRedirection(str);
            ls.suivant();
        }//if
        else
            cerr << "erreur de syntaxe caractere "
                << ls.getColonne() << endl;

        break;

    }//switch
} //while

if (ls.getSymCour().getCode() == BACKGROUNDDED)
{
    cmd->setBackground();
    ls.suivant();
} //if

return cmd;
} //getSimpleCommand

```

```

////////////////////////////////////
Parseur::Parseur(const char* str, bool file)
    : ls(str, file) { }

////////////////////////////////////
Command *Parseur::parse()
{
    Command* cmd = getSimpleCommand();
    if (cmd == NULL && ls.getSymCour().getCode() != FIN)
    {
        cerr << "erreur de syntaxe caractere " << ls.getColonne()
            << endl;
        return NULL;
    }//if
    while (ls.getSymCour().getCode() != FIN)
    {
        //test si le symbole courant n'est pas le dernier
        if (ls.getSymCour().getCode() == PIPE)
        {
            ls.suivant();
            cmd = new PipeCmdLinker(cmd, getSimpleCommand());
        }//if
        else if (ls.getSymCour().getCode() == POINT_VIRGULE)
        {
            ls.suivant();
            cmd = new SemiColumnCmdLinker(cmd, getSimpleCommand());
        }//if
        else
        {
            cerr << "erreur de syntaxe caractere "
                << ls.getColonne() << endl;
            return NULL;
        }//else
    }//while
    return cmd;
} //parse

```

```

/*
 *
 * @File: Process.cxx
 *
 *
 * @Date: 23/12/05
 *
 *
 * @Authors: Castille, Mattio, Lichnowski
 *
 *
 */
#include <sys/types.h>
#include <signal.h>
#include <vector>

#include <iostream>
using namespace std;

```

```

#include "Process.h"
#include "nsSysteme.h"

using namespace nsSysteme;

std::vector<Process*> Process::TableProcess;

Process::Process(char **argv, pid_t pgrp, bool bg)
    : m_pid (-1), m_pgrp(pgrp), m_completed(false),
      m_stopped(false), m_status(0), m_background(bg)
{
    this->m_argv = argv;
    TableProcess.push_back(this);
} //Process

void Process::launch(int infd, int outfd, int errfd)
{
    this->m_pid = getpid();

    Signal (SIGINT, SIG_DFL);
    Signal (SIGQUIT, SIG_DFL);
    Signal (SIGTSTP, SIG_DFL);
    Signal (SIGTTIN, SIG_DFL);
    Signal (SIGTTOU, SIG_DFL);
    Signal (SIGCHLD, SIG_DFL);

    if (this->m_pgrp == 0)
        this->m_pgrp = m_pid;
    setpgid(m_pid, this->m_pgrp);

    if (infd != -1)
    {
        Dup2(infd, STDIN_FILENO);
        Close(infd);
    } //if

    if (outfd != -1)
    {
        Dup2(outfd, STDOUT_FILENO);
        Close(outfd);
    } //if

    if (errfd != -1)
    {
        Dup2(errfd, STDERR_FILENO);
        Close(errfd);
    } //if

    execvp(m_argv[0], m_argv);
    throw CExcFctSyst (string ("execvp () - Commande ") + m_argv[0]);
} //launch()

pid_t Process::getPID()
{
    return this->m_pid;
} //getPID()

```

```

bool Process::isBackground()
{
    return this->m_background;
} //isBackground

void Process::setCompleted()
{
    this->m_completed = true;
} //setCompleted

void Process::setForeground(bool continuing)
{
    tcsetpgrp(STDIN_FILENO, this->m_pgrp);
    this->m_background = false;
    if (continuing)
    {
        kill(-this->m_pgrp, SIGCONT);
    } //if

    Waitpid(this->m_pid, NULL, WUNTRACED);
    tcsetpgrp(STDIN_FILENO, getpid());
} //setForeground()

void Process::setBackground(bool continuing)
{
    this->m_background = true;
    if (continuing)
        kill(-this->m_pgrp, SIGCONT);
} //setBackground()

ostream & operator << (ostream & cout, Process p)
{
    register unsigned int i = 0;
    while (p.m_argv[i] != NULL)
    {
        cout << p.m_argv[i];
        i++;
        if (p.m_argv[i] != NULL)
            cout << " ";
    } //while
    return cout;
}

Process* Process::getProcessFromPID(pid_t pid)
{
    unsigned i;
    for (i = 0; i < TableProcess.size(); i++)
        if (TableProcess.at(i)->getPID() == pid) break;
    if (i < TableProcess.size())
        return TableProcess.at(i);
    else
    {
        cerr << "Aucun PID correspondant au numéro: " << pid << endl;
        return NULL;
    } //else
} //getProcessFromPID

int Process::getProcessIndexFromPID(pid_t pid)
{
    unsigned i;

```

```

    for (i = 0; i < TableProcess.size(); i++)
        if (TableProcess.at(i)->getPID() == pid) break;
    if (i < TableProcess.size())
        return i;
    else
    {
        cerr << "aucun PID correspondant au numéro : " << pid << endl;
        return -1;
    }//else
} //getProcessIndexFromPID

```

```

Process* Process::getProcessAt(unsigned int pos)
{
    if (pos < TableProcess.size())
        return TableProcess.at(pos);
    else
        return NULL;
}

```

```

} //getProcessFromPID

```

```

Process* Process::getLastProcess()
{
    return TableProcess[TableProcess.size() - 1];
} //getProcessFromPID

```

```

void Process::deleteLastProcess()
{
    delete TableProcess[TableProcess.size() - 1];
    TableProcess.pop_back();
} //deleteLastProcess()

```

```

void Process::deleteProcessAt(unsigned int pos)
{
    delete TableProcess[pos];
    for (register unsigned int i = pos; i < TableProcess.size() - 1; i++)
        TableProcess[i] = TableProcess[i + 1];
    TableProcess.resize(TableProcess.size() - 1);
} //deleteProcessAt()

```

```

/*
 *
 * @File: Symbole.cxx
 *
 *
 * @Date: 23/12/05
 *
 *
 * @Authors: Castille, Mattio, Lichnowski
 *
 *
 */

#include <string.h>
#include <ctype.h>
#include <iomanip>

#include "Symbole.h"
using namespace std;

/////////////////////////////////////////////////////////////////
// déclaration des variables de classe (variables "static")
const unsigned int Symbole::NB_SYMBLES;
const unsigned int Symbole::TAILLE_MAX_CHAINE;
const unsigned int Symbole::TAILLE_MAX_CHAINE_CODE;

const char Symbole::motCle[NB_SYMBLES][TAILLE_MAX_CHAINE_CODE] =
    {"", "", "", "|", "&", ";", "<", ">", ">>"};

const char Symbole::codeEnClair[NB_SYMBLES][15] =
    {"INDEFINI", "FIN", "CHAINE", "PIPE", "CAND", "POINT_VIRGULE",
    "INFERIEUR",
    "SUPERIEUR", "SUPERIEUR_DBL"};

/////////////////////////////////////////////////////////////////
Symbole::Symbole (char ch[])
{
    strncpy(chaine,ch,TAILLE_MAX_CHAINE - 1);
    if (strlen(chaine) == 0) code = FIN;
    else
        {
            register unsigned short i = 0;
            while (i < NB_SYMBLES && strcmp(chaine, motCle[i]) != 0) i++;
            if (i < NB_SYMBLES) code = i;
            else code = CHAINE;
        }
} //else
} //Symbole

/////////////////////////////////////////////////////////////////
Symbole::Symbole (unsigned short code)
{
    this->code = code;
    strcpy(chaine, motCle[code]);
} //Symbole

```

```

////////////////////////////////////
// Attention : cette fonction (opérateur) n'est pas membre de la classe
Symbole
ostream & operator << (ostream & cout, Symbole symb)
{
    cout << "Symbole " << setw(Symbole::TAILLE_MAX_CHAINE_CODE)
        << Symbole::codeEnClair[symb.code]
        << " - Code=" << setw(2) << symb.code
        << " - Chaîne=\"" << setw((strlen(symb.chaine))) << symb.chaine
        << "\"";
    return cout;
} //operator <<

```

Dirshell-a

```

/*
 *
 * @File: testCommand.cxx
 *
 *
 * @Date: 23/12/05
 *
 *
 * @Authors: Castille, Mattio, Lichnowski
 *
 *
 */

#include "Command.h"
#include "ppal.h"

#include <stdlib.h>

using namespace std;

void testSimpleCommandBuilder() {
    char *args[] = {"ls", NULL};
    char *argcd[] = {"cd", "..", NULL};

    SimpleCommand * cmd = SimpleCommand::buildCommand(2, argcd);
    cmd->execute();
    cmd = SimpleCommand::buildCommand(1, args);
    cmd->execute();
}

int std::ppal (int argc, char * argv []) throw (exception)
{
    char *arg1[] = {"ls", NULL};
    char *arg2[] = {"grep", "Lec", NULL};
    //ExternalCmd *Cmd = new ExternalCmd(3, arg);
    //InternalCmd *Cmd = new InternalCmd(1, arg);
    Command *CCmd = new ExternalCmd(1, arg1);

```

```

SimpleCommand *SCmd = new ExternalCmd(2, arg2);
CommandLinker *Cmd = new PipeCmdLinker(CCcmd, SCmd);

//Cmd->print();

Cmd->execute();
testSimpleCommandBuilder();

return 0;
} // ppal()

/*
 *
 * @File: testLecteurCaractere.cxx
 *
 *
 * @Date: 23/12/05
 *
 *
 * @Authors: Castille, Mattio, Lichnowski
 *
 *
 */

#include "LecteurCaractere.h"
#include "ppal.h"

#include <assert.h>
#include <stdlib.h>

using namespace std;

int std::ppal (int argc, char * argv []) throw (exception)
{
////////// simple test ... //////////////////////////////////////////
LecteurCaractere *lc = new LecteurCaractereString("abcd");
assert(lc->getCarCour() == 'a');
lc->suivant();
assert(lc->getCarCour() == 'b');
lc->suivant();
assert(lc->getCarCour() == 'c');
lc->suivant();
assert(lc->getCarCour() == 'd');
lc->suivant();
assert(lc->getCarCour() == EOF);
lc->suivant();
assert(lc->getCarCour() == EOF);
delete lc;
//////////

return 0;
} // ppal()

```



```

////////////////////////////////////
{
    Parseur *p = new Parseur("ls -a > testfile", false);
    Command *cmd = p->parse();
    cmd->print();
    cmd->execute();
    delete cmd;
    delete p;
}
////////////////////////////////////
////////////////////////////////////
//////////////////////////////////// another simple test ... //////////////////////////////////
{
    Parseur *p = new Parseur("ls -l >> testfile", false);
    Command *cmd = p->parse();
    cmd->print();
    cmd->execute();
    delete cmd;
    delete p;
}
////////////////////////////////////

return 0;

} // ppal()

```

```

/*
 *
 * @File: shell-a.cxx
 *
 * @Date: 15/01/06
 *
 * @Authors: Castille, Lichnowski, Mattio
 *
 */

```

```

#include <iostream>
#include <string>
#include <exception>
#include <pwd.h>
#include <sys/types.h>          // pid_t

#include "ppal.h"
#include "CstCodErr.h"
#include "CException.h"
#include "CExcFctSyst.h"
#include "nsSysteme.h"         // Sigaction()
#include "Parseur.h"
#include "Command.h"
#include "Process.h"

using namespace nsUtil;       // CException
using namespace nsSysteme;    // Fork(), Waitpid()
using namespace std;

```

```

static pid_t shell_pgid;
static string Com;

void sigchldHandler(int sig, siginfo_t *info, void *c)
{
    switch(sig)
    {
        case SIGCHLD:
        {
            int status = info->si_status;
            int pos = Process::getProcessIndexFromPID(info->si_pid);
            Process *p = Process::getProcessAt(pos);
            if (p->isBackground() && !WIFSTOPPED(status))
                Waitpid(info->si_pid, &status);

            if (WIFEXITED(info->si_status))
            {
                if (p->isBackground() ||
                    WEXITSTATUS(info->si_status) != 0)
                    cout << "[" << pos
                        << "]" (" << *p
                            << ") terminé avec la valeur "
                            << WEXITSTATUS(info->si_status)
                            << endl;
            }//if

            if (WIFSIGNALED(info->si_status))
            {
                cout << "[" << pos << "]" ("
                    << *p << ") a reçu le signal "
                    << WTERMSIG(info->si_status)
                    << " : "
                    << strsignal(WTERMSIG(info->si_status))
                    << endl;

                Process::deleteProcessAt(pos);
            }//if

            if (WIFSTOPPED(info->si_status))
            {
                cout << "[" << pos << "]" (" << *p
                    << ") stoppé avec le signal "
                    << WSTOPSIG(info->si_status) << " : "
                    << strsignal(WSTOPSIG(info->si_status))
                    << endl;
            }//if

            if (!WIFSTOPPED(info->si_status) && pos > 0)
                Process::deleteProcessAt(pos);

            break;
        }
        case SIGTERM:
            break;

        case SIGINT:
            break;

        default:
            break;
    }//switch
}//sigchldHandler

```

```

int std::ppal(int argc, char * argv []) throw (exception)
{
    if (argc != 1)
        throw CException (string ("Usage: ") + argv[0], CstExcArg);

    shell_pgid = getpid();
    if (setpgid (shell_pgid, shell_pgid) < 0)
    {
        cerr << "Ne peut mettre le shell dans son propre groupe de
processus !\n";
        exit (1);
    }//if
    tcsetpgrp(STDIN_FILENO, shell_pgid);

    struct sigaction act;
    act.sa_flags = SA_SIGINFO | SA_RESTART;
    act.sa_sigaction = &sigchldHandler;

    Sigaction(SIGINT,&act,NULL);
    Sigaction(SIGQUIT,&act,NULL);
    Sigaction(SIGTERM,&act,NULL);
    Sigaction(SIGTSTP,&act,NULL);
    Sigaction(SIGCHLD,&act,NULL);

    struct passwd * Pass;
    Pass = getpwuid(getuid());

    while (1)
    {
        cout << (*Pass).pw_name << " >";
        getline (cin, Com);
        Parseur *Par = new Parseur (Com.c_str(), false);
        Command *Cmd = Par->parse();
        if (Cmd != NULL)
        {
            try
            {
                pid_t pgrp = 0;
                Cmd->execute(pgrp);
                delete Cmd;
                delete Par;
            }//try
            catch (CExcFct & E)
            {
                if (errno == EACCES)
                {
                    cerr << "Problème d'ouverture de fichier!\n"
                    << "Verifiez les droits et la presence du
fichier.\n";

                    else if (errno == ENOENT)
                        cerr << "Fichier ou repertoire inexistant!\n";
                    else throw E;
                }//catch
            }//if
        }
    }//while
    return 0;
} //ppal

```

```
##**
##*
##* @File : Makefile
##*
##* @Authors : D. Mathieu
##*           M. Laporte
##*
##* @Date : 10/08/2000
##*
##* @Author : A. B. Dragut
##* @Version : V1.0
##*
##* @Synopsis : Makefile général
##*
##**/
include ../include/INCLUDE_H

nom = shell-a

#
COMPILER = g++ -g -c -I$(INCLUDE) -Wall -Werror $*.cxx
#
#
# Edition de liens
#
$(nom) : cible $(nom).o
        g++ -s -o $(nom).run $(nom).o -L$(UTIL) -lSys

#
$(nom).o : $(nom).cxx $(NSUTIL_H) $(CEXCEPTION_H) $(NSSYSTEME_H)
        $(COMPILER)
#
#
cible :
        cd $(UTIL); make -f MakeUtil general
#
# Nettoyage du répertoire courant : exécutable et fichiers .o
#
clean :
        clear; rm -f *.o *.run *~ -v; cd $(UTIL); make -f MakeUtil clean; rm
-f ../include/*~;
```