Licence d'informatique
Module de C/Unix

La commande `fact`

Examen – Parties Unix et C

Philippe MARQUET

Janvier 2004

Les documents de cours et TD sont autorisés.

Ce sujet propose de développer une solution en deux parties : une partie écrite en langage C et une partie écrite en Bourne-shell.
On rendra deux copies séparées pour la partie langage C et pour la partie Unix.
La partie C (questions 5 à 8) est à rendre sur une *copie blanche*; la partie Unix (questions 1 à 4) sur une *copie de couleur*.
On introduira tout code C ou Bourne-shell par une brève description de son comportement; on attachera un commentaire à toute portion non triviale de code.

Nous allons considérer la facturation de l'utilisation des ressources d'une machine Unix. Une liste de travaux (des commandes) va être exécutée. Les ressources utilisées pour chacune de ces exécutions seront identifiées. À partir de ces ressources un coût sera calculé. Ces traitements seront réalisés par la commande `fact` qui sera développée.

Un travail = exécution d'une commande + facturation

Un travail à exécuter est identifié par les paramètres suivants :

`COMMAND` le nom d'une commande Unix devant être exécuter, avec ses options et paramètres;

`DIRECTORY` le répertoire absolu dans lequel l'exécution de la commande doit avoir lieu;

`OUTPUT` le nom d'un ficher dans lequel les affichages de l'exécution de la commande sont redirigés;

`USERNAME` le nom de connexion (login) de l'utilisateur pour le compte duquel la commande sera exécutée;

`FORMULA` le nom de la formule de calcul du coût qui sera imputé à l'exécution de la commande en fonction des ressources utilisées.

La facturation de l'exécution d'un travail consiste
- à se déplacer dans le répertoire `DIRECTORY`;
- à exécuter la commande `COMMAND` dont le résultat est redirigé dans `OUTPUT`;
- récupérer les ressources utilisées par cette exécution; nous allons pour ce faire utiliser la commande `time` décrite ci-dessous;
- calculer le coût de l'utilisation de ces ressources en appliquant la formule identifiée par `FORMULA`; cette application des formules est décrite ci-dessous.

Identification des ressources

La commande `time` (en fait la commande `/usr/bin/time`) exécute une commande donnée en paramètre et évalue les ressources employées par son exécution. L'option `-f` de `time` précise le format dans lequel le résultat doit être affiché. Les conversions suivantes seront utilisées :

%U désigne le nombre de secondes CPU passées par le processus en mode utilisateur ;

%S désigne le nombre de secondes CPU passées par le processus en mode noyau ;

%I désigne le nombre de lectures depuis un fichier ;

%O désigne le nombre d'écritures dans un fichier.

comme dans l'exemple de session Bourne-shell

```
% format=" user: %U \n system: %S \n input: %I \n output: %O"
% /usr/bin/time -f "$format" cp -R /boot /root/boot
 user: 0.00
 system: 0.08
 input: 125
 output: 125
%
```

Application d'une formule de calcul

L'application d'une formule de calcul consiste à exécuter la commande du répertoire `$FORMULA_DIR` (variable d'environnement définie, à défaut `/var/lib/fact/formula/`) correspondant au nom de la formule en lui passant en paramètre les quatre valeurs

- nombre de secondes CPU utilisateur ;
- nombre de secondes CPU noyau ;
- nombre de lectures ;
- nombre d'écritures.

Cette commande va afficher le coût correspondant à l'utilisation de ces ressources. Par exemple :

```
% $FORMULA_DIR/h_creuses 0.00 0.08 125 125
0.0023
%
```

Liste de travaux à exécuter

Notre commande de facturation `fact` est paramétrée par un fichier de configuration. Un fichier de configuration est constitué d'une suite de sections.

Il existe une section générale et une section par travail à exécuter et facturer. La section générale est placée en début du fichier de configuration. Le début de la section relative à un travail de nom `test` est marquée par une ligne

```
[test]
```

Une section est formée de lignes de la forme

```
parameter = value
```

Les lignes vides (i.e. ne comportant que des espaces) et les lignes dont le premier caractère significatif est un dièse # sont ignorées.

Un fichier de configuration produit deux informations :

1. Une liste de travaux pouvant être exécutés et facturés.
2. Pour chacun des travaux, la valeur des paramètres l'identifiant. Pour un travail, la valeur d'un paramètre donné est définie dans la section relative au travail ; à défaut dans la section principale.

Ainsi l'exemple de fichier de configuration `fact.cfg`

```
# fact.cfg
# Fichier de configuration pour un premier test de ma commande fact

# le répertoire d'exécution par défaut
DIRECTORY = /tmp
# l'utilisateur par défaut
USERNAME = nobody
```

```
# la sortie par defaut
OUTPUT = /dev/null

# un travail pour le compte d'un utilisateur
[calcul]
COMMAND = /home/duchmol/bin/compute_pi 1000000
USERNAME = duchmol
DIRECTORY = /home/duchmol
OUTPUT = /home/duchmol/pi.txt
FORMULA = h_creuses

# un travail utilisant des paramètres par défaut
[sauvegarde]
COMMAND = /usr/local/scripts/save_users
FORMULA = root_admin
```

identifie deux travaux `calcul` et `sauvegarde`. Le travail `sauvegarde` sera exécuté dans le répertoire `/tmp` et facturé à `nobody`.

Architecture de nos développements... donc de l'examen

Nous allons développer une commande `fact` en Bourne-shell. Les commandes d'évaluation des formules du répertoire `$FORMULA_DIR` seront développées en C.

1 Partie Unix de l'examen

Il s'agit dans un premier temps de développer de petits outils sous forme de fonctions Bourne shell pour ensuite écrire la commande `fact`.

Dans la suite,

- *`cfg-file`* désigne un fichier de configuration.
- *`work`* désigne un travail.
- *`param`* désigne un paramètre de configuration.

La fonction

```
get_works cfg-file
```

affiche la liste des travaux identifiés dans le fichier *`cfg-file`*. Par exemple pour le fichier `fact.cfg` précédent :

```
% get_works fact.cfg
calcul
sauvegarde
%
```

Exercice 1 (fonction `get_works`)
Donnez un algorithme et le code Bourne shell commenté de la fonction `get_works`. ■

La fonction

```
get_val cfg-file work param
```

affiche la valeur du paramètre *`param`* pour le travail *`work`*. Par exemple :

```
% get_val fact.cfg sauvegarde FORMULA
root_admin
% get_val fact.cfg sauvegarde USERNAME
nobody
%
```

Exercice 2 (fonction `get_val`)
Donnez un algorithme et le code Bourne shell commenté de la fonction `get_val`. ■

La fonction `do_work` exécute un travail désigné et affiche le nom de ce travail, le nom de l'utilisateur et le coût correspondant à l'exécution sur la sortie standard. Par exemple

```
% do_work fact.cfg sauvegarde
[sauvegarde] nobody 0.0015
```

Exercice 3 (fonction do_work)
Donnez un algorithme et le code Bourne shell commenté de la fonction `do_work`. ■

Enfin, la commande `fact` exécute l'ensemble des travaux désignés d'un fichier de configuration donné et reporte un ensemble de lignes au même format que `do_work` :

```
fact cfg-file [work]...
```

En l'absence de paramètre *`work`*, l'ensemble des travaux du fichier *`cfg-file`* sont considérés.

Exercice 4 (commande fact)
Donnez un algorithme et le code Bourne shell commenté de la commande `fact`. ■

2 Partie C de l'examen

L'application d'une formule à une liste de variables (par exemple les temps CPU et nombres d'entrées/sorties) met en jeu des coefficients pour produire une valeur (le coût).

Les variables sont représentées par des noms dans les formules. Les coefficients sont entiers.

On suppose défini dans une bibliothèque ad hoc le type `value_t` et les fonctions de manipulation (entrée, sortie, calcul) sur ce type. L'interface de cette bibliothèque est donnée par le fichier `val.h` suivant

```
#ifndef _VAL_H_
#define _VAL_H_

typedef float value_t;

/* entrées/sorties */
void read_value(value_t *val);
void write_value(value_t val);

/* conversion depuis une chaîne de caractères */
void a_to_value(value_t *val, const char *s);

/* calcul */
void sum_value(value_t n1, value_t n2, value_t *r); /* *r = n1 + n2 */
void mul_value(value_t n, int coeff, value_t *r);   /* *r = n * c   */
void div_value(value_t n, int coeff, value_t *r);   /* *r = n / c   */

#endif
```

On ne devra jamais faire de calcul directement sur les valeurs mais systématiquement utiliser les fonctions de la bibliothèque `val`.

Une formule est caractérisée par les éléments suivants :
- le résultat d'une formule est une valeur ;
- une formule peut être la somme de deux formules ;
- une formule peut être une formule multipliée ou divisée par un coefficient ;
- une formule peut se résumer à la valeur d'une variable.

On utilisera les définitions suivantes disponibles dans le fichier `formula_t.h` :

```
#ifndef _FORMULA_T_H_
#define _FORMULA_T_H_

#define NAME_LG    16

enum op_e {SUM_OP, MUL_OP, DIV_OP, VAL_OP};
```

```
struct var_s {
    char         var_name[NAME_LG]; /* nom de la variable */
    value_t      var_val;           /* valeur de cette variable */
    struct var_s *var_next;         /* chaînage de toutes les variables */
};

struct form_elem_s {
    enum op_e                         fe_operator;
    union {
        struct var_s       *fe_var;
        struct form_elem_s *fe_form;
        int                fe_coeff;
    }                                 fe_operand1,
                                      fe_operand2;
};

#endif
```

Chaque élément de type `struct var_s` réalise l'association d'une valeur à un nom de variable. Toutes les variables utilisées dans une formule sont chaînées dans une liste en utilisant le champs `var_next`.

Le type des champs `fe_operand1` et `fe_operand2` dépend de la valeur du champ `fe_operator`, il est cohérent avec les propriétés des formules :

valeur de `fe_operator`	type de `fe_operand1`	type de `fe_operand2`
`SUM_OP`	`(struct form_elem_s *)`	`(struct form_elem_s *)`
`MUL_OP` ou `DIV_OP`	`(struct form_elem_s *)`	`int`
`VAL_OP`	`(struct var_s *)`	champ non utilisé

Chaque élément de type `struct form_elem_s` représente une formule que l'on supposera correcte. Par exemple, la formule

$$(a * 2 + b + c)/4 + c$$

peut être accessible par deux variables déclarées

```
struct form_elem_s *formule;
struct var_s *liste_vars;
```

et peut être représentée comme à la figure 1.

La fonction

```
void input_values(struct var_s *list, char *argv[]);
```

remplit les valeurs des variables de la liste `list` avec les chaînes de caractères du tableau `argv`. L'extrait de code C suivant illustre l'utilisation de cette fonction ;

```
int
main(int argc, char *argv[])
{
    struct form_elem_s *formule;
    struct var_s *liste_vars;

    /* creation de la formule et de la liste des variables */
    ...

    /* affectation des variables */
    input_values(liste_vars, argv+1);

    /* ... */
```

La commande suivante est un exemple d'utilisation du programme issu de ce code qui va affecter les trois variables de la liste :

```
% ./a.out 10.0 8.0 12.0
```

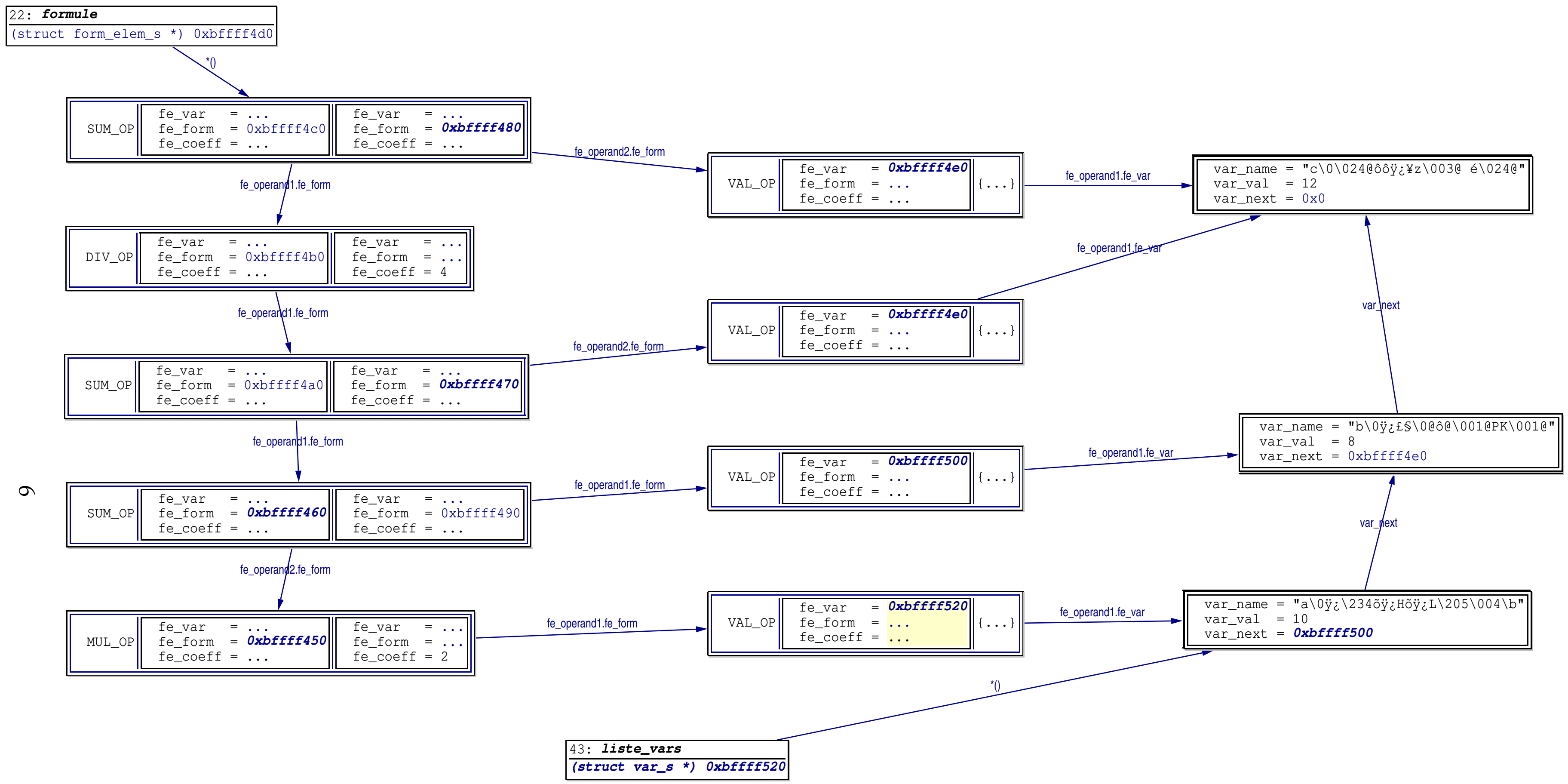

FIG. 1 – Représentation possible de la formule

Exercice 5 (fonction `input_values()`)
Donnez le principe et le code C de la fonction `input_values()`. ■

La fonction

```
void print_formula(struct form_elem_s *formula);
```

calcule la valeur d'une formule en utilisant les valeurs déjà associées aux variables; la valeur résultante de la formule est ensuite affichée.

Exercice 6 (fonction `print_formula()`)
Donnez l'algorithme et le code C de la fonction `print_formula()`. ■

On suppose que tous les éléments de type `struct var_s` et de type `struct form_elem_s` ont été alloués dynamiquement `malloc()`. La fonction

```
void free_formula(struct form_elem_s **formula,
                  struct var_s **var_list);
```

désalloue tous ces éléments et réinitialise à zéro ses paramètres.

Exercice 7 (fonction `free_formula()`)
Donnez le principe et le code C de la fonction `free_formula()`. ■

Exercice 8 (bibliothèque `formula`)
Les fonctions des exercices précédents utilisent possiblement d'autres fonctions que vous avez définies.

Donnez le source C d'un fichier qui une fois compilé séparément produit une bibliothèque qui fournit exactement (ni plus, ni moins) les trois fonctions définies par

```
#ifndef _FORMULA_H_
#define _FORMULA_H_

#include "formula_t.h"

void input_values(struct var_s *list, char *argv[]);
void print_formula(struct form_elem_s *formula);
void free_formula(struct form_elem_s **formula,
                  struct var_s **var_list);
#endif
```

Bien entendu, ne réécrivez pas entièrement le corps de vos fonctions mais abrégez en ne recopiant que l'essentiel. ■

Fin du sujet