Licence 3 informatique — C avancé
Université de Marne la Vallée — Année 2018-2019

Projet de programmation C : Profilage

L'objectif de ce projet est de développer un outil simple pour la visualisation de l'exécution d'un programme en langage C. Cet outil dont l'utilisation est incontournable en optimisation combinatoire s'appelle le profileur.

Description, objectifs et motivations

Dans la plupart des langages de programmation, l'unité atomique composant les programmes est la fonction (méthode pour les langages objets). Le profilage consiste à parcourir l'arbre des appels de fonctions (ou de méthodes...) et de collecter les temps d'exécution de chaque appel. L'outil profiler propose alors un résumé facilement exploitable pour le programmeur qui peut identifier très facilement quelles sont les parties critiques du code à optimiser.

Le profilage est complètement dépendant de la manière dont le programme a été utilisé lors de son exécution. Il dépend ainsi des arguments du programme à profiler et de toutes les interactions produites par l'utilisateur (mouvement de la souris, clic, entrées au clavier, ...). Pour le langage C, on procède de la manière suivante :

- Mise en place des balises de profilage dans le code source,
- exécution du programme et génération de données brutes comportementales du programme,
- exploitation des données brutes via un profileur et visualisation.

Un utilitaire ultra léger (relativement incomplet et non graphique) existe sous `UNIX` pour profiler le C : **gprof**. Son utilisation se déroule comme il suit.

- Compilation des sources avec l'option `-pg` du compilateur `gcc` (place les balises),
- exécution du programme (qui génère alors un fichier de données brutes `gmon.out`),
- exploitation des données brutes via le profileur `gprof`.

Profilage d'un jeu "snake" programmé l'année (2012-2013) par un étudiant de L2. Le profiler utilisé est ici **gprof** et l'utilisateur a mangé 40 proies avant de rencontrer la queue du ver. Le programme s'est terminé normalement.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.11      0.02     0.02      224     0.09     0.09  affiche_Damier
  0.00      0.02     0.00      232     0.00     0.00  est_Valide
  0.00      0.02     0.00      224     0.00     0.00  creer_Coord
  0.00      0.02     0.00      223     0.00     0.00  coordonnees_Deplacement
  0.00      0.02     0.00      223     0.00     0.00  mise_A_Jour_Jeu
  0.00      0.02     0.00      182     0.00     0.00  deplacement_Ver
  0.00      0.02     0.00       40     0.00     0.00  Ver_mange_proie
  0.00      0.02     0.00        1     0.00     0.00  affichage_victoire_defaite
  0.00      0.02     0.00        1     0.00     0.00  creer_Damier
  0.00      0.02     0.00        1     0.00     0.00  creer_Ver
  0.00      0.02     0.00        1     0.00    20.02  graphique_clavier
  0.00      0.02     0.00        1     0.00     0.00  init_Damier
  0.00      0.02     0.00        1     0.00     0.00  init_Proies
  0.00      0.02     0.00        1     0.00     0.00  initialisation_fenetre
```

Un profileur donne toujours les informations suivantes pour chaque fonction appelée :

- le nombre d'appel de la fonction
- le temps total pris par l'exécution de toutes les instances de la fonction
- le temps moyen par appel

Malheureusement, la visualisation par tableau à double entrée ne conserve pas les filiations d'appel (pile d'exécution) entre fonctions. On verra plus tard comment proposer une visualisation graphique qui permet de mieux voir qui appelle qui.

Mise en oeuvre

Par soucis de simplicité, le marquage du code à profiler sera fait à l'aide de simples macros que le préprocesseur va substituer dans toutes les fonctions. Chaque fonction commencera avec la macro `PROFILE` et il vous sera demandé de patcher le mot clé `return` du langage C à l'aide d'une macro du même nom. Ainsi, une factorielle récursive à profiler aura pour code :

```
1  #define PROFILE ... beaucoup de code
2  #define return ... beaucoup de code puis return
3
4  int fact(int n){
5    PROFILE
6    int ans;
7    if (n<=1){
8      return 1;
9    }
10   else{
11     ans = n*fact(n-1);
12     return ans;
13   }
14 }
15
16 int main(void){
```

```
17     PROFILE
18     fact(10);
19     return 1;
20  }
```

Le profilage de ce programme donne la donnée brute suivante.

```
main -- time : 1375041850.230101s
fact -- time : 1375041850.230136s
fact -- time : 1375041850.230146s
fact -- time : 1375041850.230155s
fact -- time : 1375041850.230165s
fact -- time : 1375041850.230174s
fact -- time : 1375041850.230184s
fact -- time : 1375041850.230194s
fact -- time : 1375041850.230203s
fact -- time : 1375041850.230213s
fact -- time : 1375041850.230224s
END -- time : 1375041850.230233s
END -- time : 1375041850.230243s
END -- time : 1375041850.230254s
END -- time : 1375041850.230263s
END -- time : 1375041850.230273s
END -- time : 1375041850.230282s
END -- time : 1375041850.230292s
END -- time : 1375041850.230301s
END -- time : 1375041850.230310s
END -- time : 1375041850.230320s
END -- time : 1375041850.230329s
```

Un `END` ferme systématiquement la dernière fonction lancée non terminée. Nous rappelons que la macro `__FUNCTION__` est substituée par le préprocesseur par le nom (chaîne de caractères) de la fonction étant exécutée.

À chaque entrée dans une fonction, son nom ainsi qu'une date absolue en seconde sont rajoutés dans le fichier de données brutes. À chaque fois qu'on sort d'une fonction, un "END" ainsi qu'une date absolue en seconde sont aussi rajoutés dans les données brutes. Pour avoir une date précise et absolue, utilisez la fonction `clock_gettime` (section 2 du manuel `Unix`).

Ces données brutes devront être sauvegardées dans un fichier `profile.log`. Ces données devront OBLIGATOIREMENT être sauvegardées comme plus haut. Le format est fondamental dans le sens que vous devrez produire un profileur capable de profiler les données venant des autres étudiants tout comme les autres étudiants devront aussi être capables d'exploiter les données brutes venant de votre profileur.

Ainsi les débuts d'exécution ressembleront comme au dessus à :
`non_de_fonction -- time :  <un_double>s`
(nom de la fonction, espace, dash, dash, espace, time, espace, deux points, espace, temps absolu en secondes, la lettre s)
et les fins de fonctions à :
`END -- time :  <un_double>s`
(END, espace, dash, dash, espace, time, espace, deux points, espace, temps absolu en secondes, la lettre s)

Pour garantir ce comportement agréable vis à vis des macros `PROFILE` et `return`, nous ferons les deux suppositions suivantes (quitte à modifier les sources à profiler):

- Toutes les fonctions se terminent par le mot clé `return`. Les fonctions retournant le type `void` se terminent donc par `return ;` (c'est à dire retourner rien du tout).
- Aucun appel de fonction n'est autorisé après le mot clé `return`. Tous les sous-appels de fonction se font dans le corps des fonctions et pas en fin. Ainsi, une factorielle récursive ne se termine pas par `return n*fact(n-1)` mais par deux lignes où la première fait l'appel récursif et stocke le résultat dans une variable locale et une seconde ligne où un `return` retourne la valeur calculée la ligne précédente.

La récupération du temps devra aller jusqu'à l'ordre de la microseconde (10^{-6} seconde). Une solution possible et raisonnable est d'utiliser la fonction `clock_gettime` de la bibliothèque <`time.h`> (attention, l'utilisation de la fonction nécessite un linkage "-lrt" pour le compilateur `gcc`).

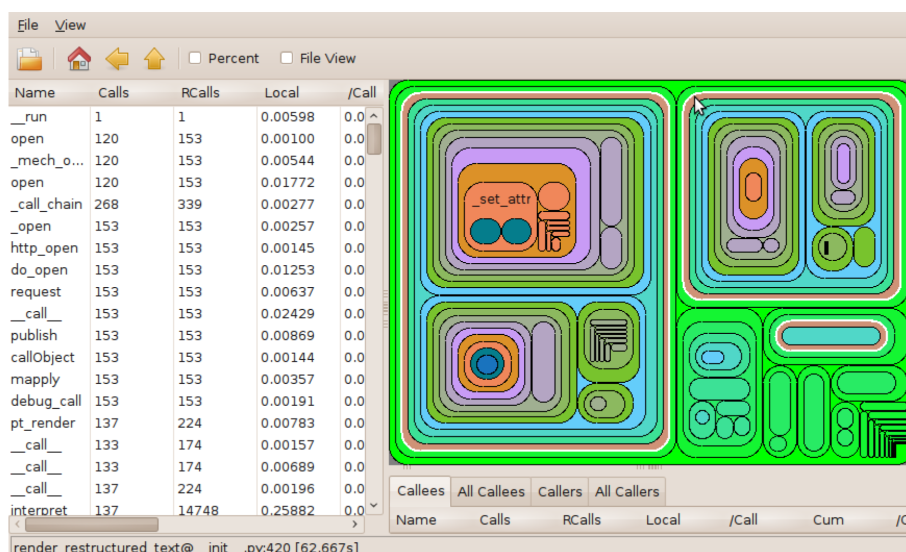
Une fois les données brutes automatiquement générées lors de l'exécution du programme, il faut maintenant les exploiter. Une première remarque consiste à remarquer que les données brutes (non de fonction + donnée temporelle) dans leur lecture chronologique correspondent exactement à la lecture profondeur préfixe de l'arbre des appels des différentes fonctions appelées lors de l'exécution du programme.

À partir de données brutes, le projet consiste maintenant à proposer un utilitaire programmé en langage C pour exploiter les données. À partir du log généré par les macros et en utilisant une structure (probablement chaînée : les arbres fils gauche frère droit semblent parfaits pour cela) adaptée, votre programme devra reconstruire l'arbre des appels, ce dernier ayant pour racine le premier appel de la fonction `main`. Chaque noeud correspondra à un appel de fonction et comme les données présentent aussi les temps d'entrées et de sorties des fonctions, vous devrez enregistrer le temps d'exécution pour chaque appel.

Une fois l'arbre construit, l'enjeu est de proposer une ou plusieurs visualisations graphiques associées au `log` de profilage.

Représentation graphique

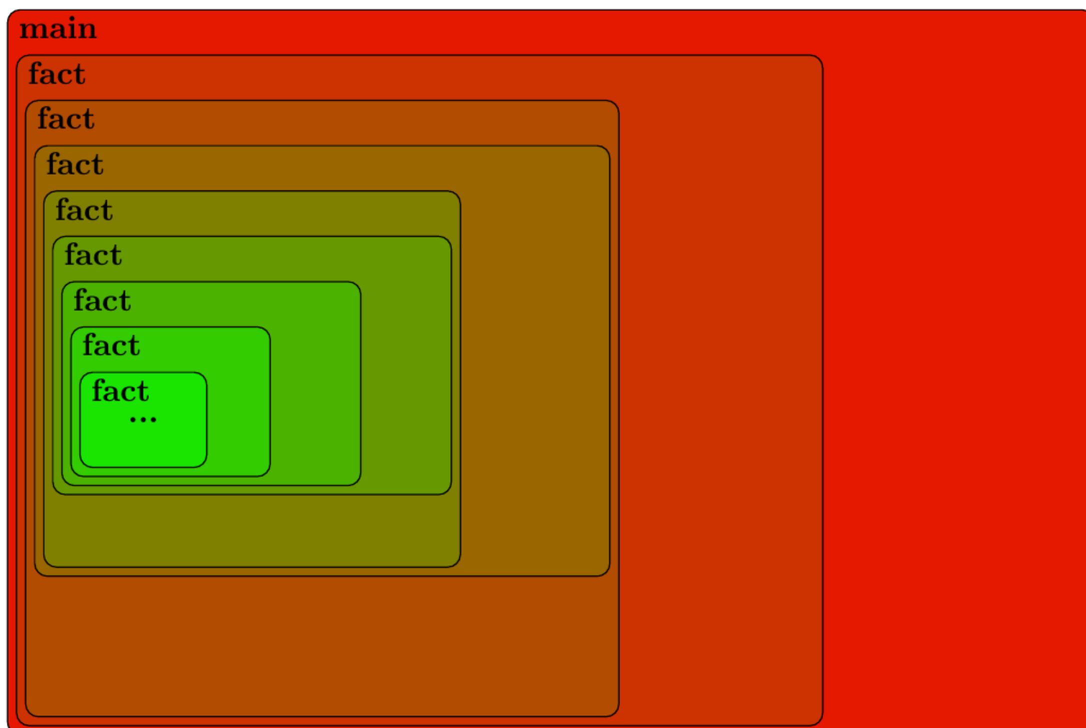
L'utilitaire Python **RunSnake** propose une visualisation agréable du profilage.



L'idée est simple et efficace, on associe une couleur (de manière automatique) à chaque appel de fonction. Une solution raisonnable consiste à faire un dégradé (fonction affine(droite) des trois variables r,g et b) du vert au rouge (0% pour le vert et 100% pour le rouge) et à colorier chaque zone suivant son temps d'exécution. Il faut aussi penser à laisser un peu de place pour que l'oeil de l'utilisateur délimite les appels, sous-appels mais aussi pour placer un nom de fonction permettant d'identifier l'appel associé à la zone.

On commence en partant d'un grand rectangle pour le premier appel de `main`, ce dernier doit occuper une place raisonnable sur l'écran (penser aux petites résolutions d'écran). Ensuite, pour toute fonction, si le rectangle est plus large que haut, on le coupe verticalement et horizontalement si le rectangle est plus haut que large.

Voici une idée de ce que l'on peut produire pour le calcul de la factorielle de 10 :



Lorsqu'il n'est plus raisonnable de descendre dans l'arbre des appels (les rectangles deviendraient trop petits pour écrire un nom de fonction par exemple), on peut tronquer en indiquant que la subdivision continue.

Conditions de développement

Le but de ce projet est moins de pondre du code que de développer le plus proprement possible. C'est pourquoi vous développerez ce projet en utilisant un système de gestion de versions git. Le serveur à disposition des étudiants de l'Université est disponible à cette adresse : `https://forge-etud.u-pem.fr/`. Comme nous attendons de vous que vous le fassiez sérieusement, votre rendu devra contenir un dump du fichier de logs des opérations effectuées sur votre projet, extrait depuis le serveur de gestion de versions git, afin que nous puissions nous assurer que vous avez bien développé par petites touches successives et propres (commits bien commentés), et non pas avec un seul commit du résultat la veille du rendu. L'évaluation tient compte de la capacité du groupe à se diviser équitablement le travail.

Remarques importantes :

- L'intégralité de votre application doit être développée exclusivement en langage C (la documentation technique dynamique fait évidemment exception). Toute utilisation de code dans un autre langage (y compris C++) vaudra ZÉRO pour l'intégralité du projet concerné.
- En dehors des bibliothèques standards du langage C, il est interdit d'utiliser du code externe : vous devrez tout coder vous-même.
  Toute utilisation de code non développé par vous-même vaudra ZÉRO pour l'intégralité du projet concerné.
- Tout code commun à plusieurs projets vaudra ZÉRO pour l'intégralité des projets concernés.

Conditions de rendu :

Vous travaillerez en binôme et vous lirez avec attention la Charte des Projets. Il faudra rendre au final une archive `tar.gz` de tout votre projet (tout le contenu de votre projet `svn`), ce dernier contenant les balises de profilage, des conseils pour faciliter leur insertion dans le code à profiler, des exemples de codes prêts à être profilés, des exemples de données brutes (notamment celles que vous avez mis à disposition des autres groupes), les sources de votre application de visualisation et ses moyens de compilation. Il sera alors crucial de lire des recommandations et conseils d'utilisation de `svn` sur la plate-forme `moodle` (à venir). Vous devrez aussi donner des droits d'accès à votre chargé de TD à votre projet via l'interface `redmine`.

Un exécutable `myprofiler` devra alors être produit et doit pouvoir permettre l'exploitation de données de profilage. Naturellement, toutes les options que vous proposerez (ne serait-ce que –help) devront être gérées avec `getopt` et getopt_long.

La cible `clean` doit fonctionner correctement. Les sources doivent être propres, dans la langue de votre choix, et commentées. C'est bien de se mettre un peu à l'anglais si possible.

Votre archive devra aussi contenir :

- Un fichier `log_dev` correspondant au dump des logs de votre projet (nom des commits, qui? et quand?), extrait depuis le serveur de gestion de versions que vous aurez utilisé.
- Un fichier `makefile` contenant les règles de compilation pour votre application ainsi que tout autre petit bout de code nécessitant compilation (comme les tests par exemple).
- Un dossier `doc` contenant la documentation technique de votre projet ainsi qu'un fichier `rapport.pdf` contenant votre rapport qui devra décrire votre travail. Si votre projet ne fonctionne pas complètement, vous devrez en décrire les bugs.
- Un dossier `src` contenant les sources de votre application.
- Un dossier `include` contenant tous les headers de vos différents modules. Ici devra se trouver un fichier `macro_profiler.h` contenant vos macros de génération des données brutes avec sécurisation contre les inclusions multiples à l'aide d'une macro `__MACRO__PROFILAGE__`.

- Un dossier `bin` contenant à la fois les fichiers objets générés par la compilation séparée mais aussi votre exécutable généré par le `makefile` nommé `myprofiler`.
- Aucun fichier polluant du type `bla.c~` ou `.%smurf.h%` généré par les éditeurs. Votre dossier doit être propre !
- Si possible, la partie html générée par l'utilitaire `doxygen` à partir de votre application de visualisation. Ceci est optionnel mais tellement plus propre.

Sachant que de nombreux vilains robots vont analyser et corriger votre rendu avant l'oeil humain, le non respect des précédentes règles peuvent rapidement avorter la correction de votre projet. Le respect du format des données est une des choses des plus critiques.

Niveau de rendu sergent de police

Le minimum vital pour un étudiant de L3.

- Implantation des macros de récupération des données brutes.
- Reconstitution de l'arbre des appels via la lecture du fichier de données brutes.
- Affichage d'un tableau résumant l'exécution du programme dont les données sont obtenues en traitant l'arbre des appels.
- Exploitation d'un fichier de `log` d'un autre groupe (preuve à l'appuie et noms des personnes composants le binôme).
- Un autre groupe a pu exploiter vos données brutes (preuve à l'appuie et noms des personnes composants le binôme).

Niveau de rendu lieutenant de police

- Toutes les fonctionnalités du niveau de rendu sergent de police.
- Visualisation graphique des résultats via une interface graphique programmée avec la `libMLV`.
- Fonctions de tris sur le résumé des appels. Le tableau récapitulatif peut ainsi être trié au moins par temps, par nombre d'appels, par temps cumulé.
- Être capable de profiler la factorielle donnée en exemple plus haut, de rajouter les balises de profilage sur votre profileur et enfin de profiler votre profileur qui profile la factorielle de 10 (c'est le serpent qui se mord la queue mais cela permet de voir l'efficacité de votre profileur).
- Utilisation de `doxygen` obligatoire pour la documentation de votre projet.

Niveau de rendu Columbo

"C'est tout un programme me disait ma femme..." *(Peter Falk dans Columbo)*

- Toutes les fonctionnalités du niveau de rendu lieutenant de police.
- Possibilité de parcourir l'arbre des appels et donc de se promener dans l'exécution. Par défaut, on profile à partir du premier appel de la fonction `main`, maintenant l'utilisateur peut choisir une autre racine pour profiler un sous-arbre de l'arbre total des appels. Le tableau récapitulatif ainsi que la représentation graphique devront être recalculés dynamiquement.
- Réfléchir (quitte à proposer un prototype, c'est à dire un nouvel exécutable) au marquage automatique d'un projet. Autrement dit, comment rajouter automatiquement les balises de profilage ?

Pour aller plus loin...

Tous les langages sérieux possèdent un outil de profilage (**gprof** en C, **%prun** et **Runsanke** en Python, **jvmmonitor** pour Java, PHP, **ocamlprof** pour OCaml, ...).

En faisant quelques recherches sur internet, vous pourriez voir quelles sont les fonctionnalités importantes d'un profileur et comment on les utilise dans des cas pratiques où la complexité du code est cruciale.

Toute amélioration intéressante sur votre projet sera un plus dans son évaluation.