

Comment mesurer les temps de calcul de vos programmes ?

Bonne pratique et pièges à éviter

Caroline Prodhon (carolineprodhon@hotmail.com)

Philippe Lacomme (placomme@isima.fr)

Ren Libo (ren@isima.fr)

Remerciements à :

Christophe Gouinaud,
et Antoine Mahul.

Modification du 21/05/2014
sur le code Visual Studio C++ :
Maxime Chassaing (chassain@isima.fr)

Présentation des problèmes

Un programme se caractérise par deux informations très importantes :

- le **User Time** : temps d'exécution pour l'utilisateur ;
- le **CPU Time** : temps qu'un programme à passer sur le processeur.

Le **User Time** est fonction de la charge de calcul de votre machine : elle fluctue. Un programme fonctionnant seul sur une machine peut avoir un temps d'exécution de 10 min alors que le temps d'exécution passera à 14 min si la machine est utilisée à surfer sur internet pendant le déroulement des calculs.

Le **CPU Time** est le temps passé par un programme sur le processeur. Ce temps est une constante : il ne dépend pas de la charge de travail de votre machine.

Lorsqu'on mesure les performances d'un algorithme de RO on a tendance à mesurer le User Time alors que cela est une erreur surtout si on travaille sur une station de travail multi-utilisateurs. Sous Windows, dès que la machine est sollicitée (ouverture de fichier etc.) à des tâches annexes (dès qu'on utilise un peu le PC sur lequel le programme tourne) le User Time sera très différent du CPU Time et ceci dans des proportions importantes : surfer sur internet pendant les calculs impacte de 6-10% le User Time.

Conclusion :

Il est absolument nécessaire de travailler avec le CPU Time. Nous allons présenter les solutions existantes en Pascal et en C sous Windows et Linux. Chaque système nécessite une solution qui lui est propre.

	Windows	Linux
Pascal	Section 1.	Section 3
C	Section 4	Section 2

1. Pascal - Windows

1.1. Réalisation du programme

Code : Pascal_Windows.rar

Téléchargement : http://www.isima.fr/~lacomme/temps/Pascal_Windows.rar

Unités utilisées :

UcpuUsage.pas

Unit1.pas

UcpuUsage.pas est unité utilisée pour calculer le taux d'utilisation processeur.

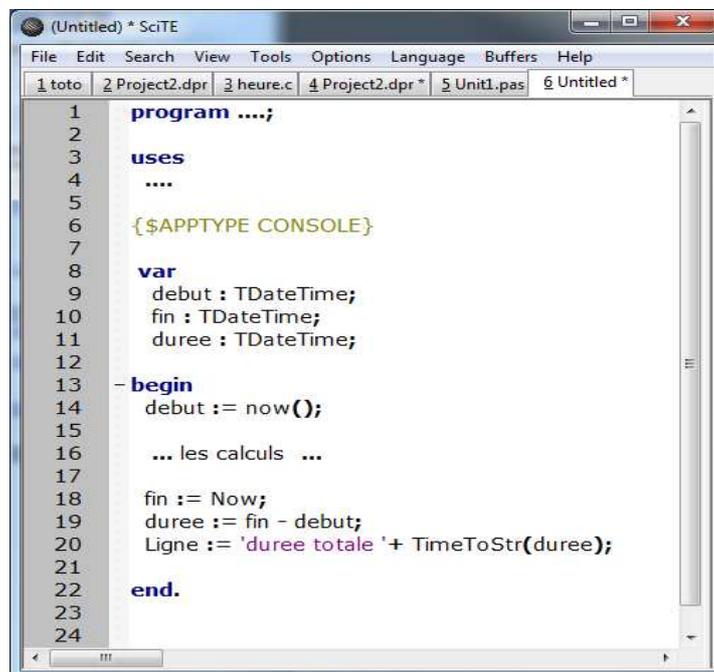
Unit1.pas sert à calculer le CPU Time.

Le programme implémente 4 méthodes :

- la méthode 1 → utilise GetCPUTick
- la méthode 2 → utilise Now() et TdateTime
- la méthode 3 → utilise GetTickCount.

Remarque :

Habituellement, on utilise tous **Now** et nos programmes sont de la forme :



```
1  program ....;
2
3  uses
4  ....
5
6  {$APPTYPE CONSOLE}
7
8  var
9  debut : TDateTime;
10 fin : TDateTime;
11 duree : TDateTime;
12
13 - begin
14  debut := now();
15
16  ... les calculs ...
17
18  fin := Now;
19  duree := fin - debut;
20  Ligne := 'duree totale ' + TimeToStr(duree);
21
22  end.
23
24
```

Figure 1. Programme Delphi usuel

Cette manière de procéder conduit à mesurer le User Time et non le CPU Time.

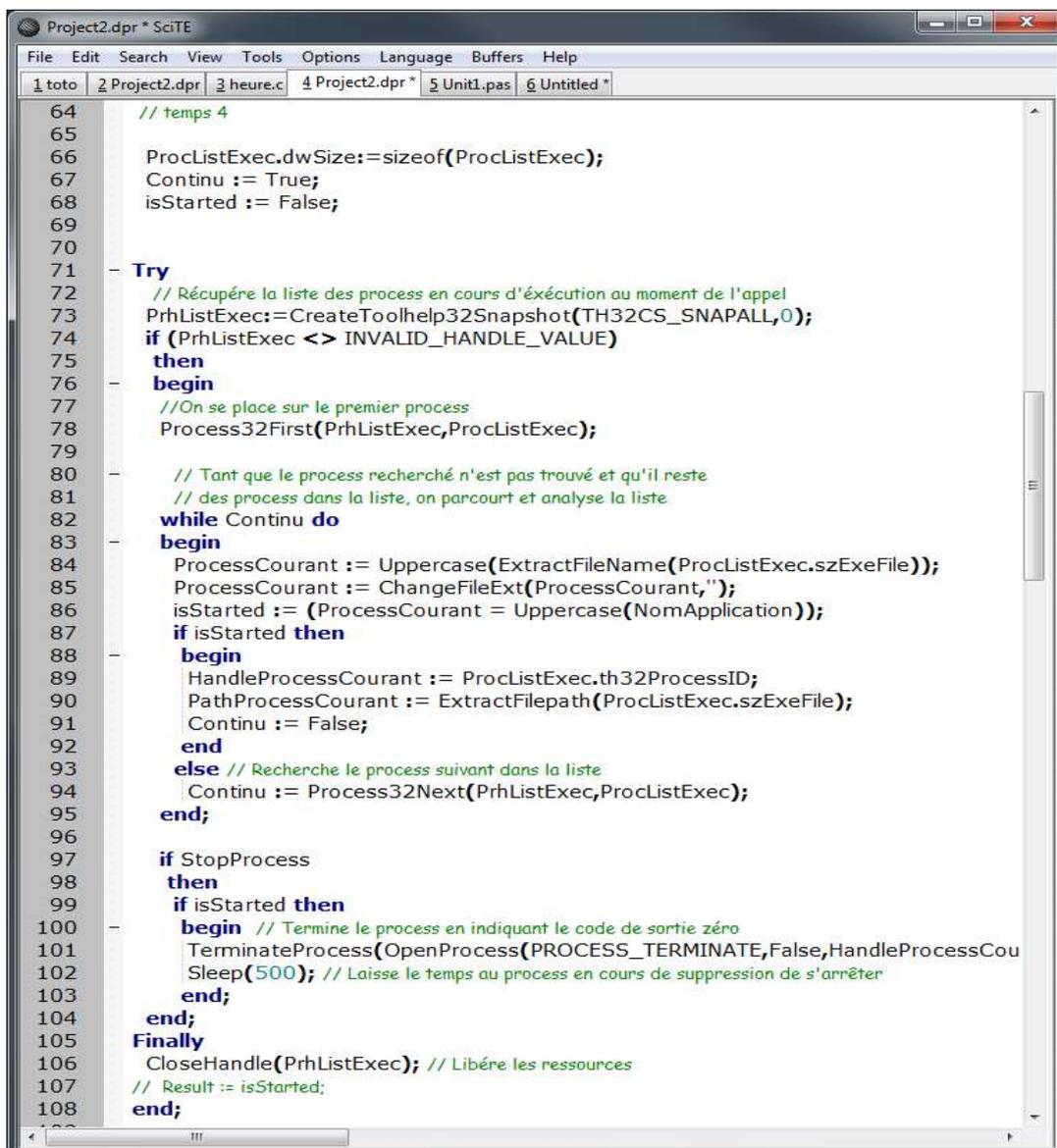
Mesurer le CPU Time :

Le principe est un peu complexe. Il s'agit de procéder comme suit :

- accéder à la liste des processus tournant sous Windows ;
- trouver dans la liste le processus pour lequel on veut mesurer le CPU Time ;
- effectuer la mesure.

Parcourir la liste des processus se fait dans le Try... Finally. Notons que pour le cas qui nous intéresse :

- le StopProcess (on arrête pas le process qu'on est en train de mesurer) ;
- il faut rechercher le process de nom PROJECT2 en majuscule et sans extension. Cela est indiqué au début du programme par : **NomApplication:='PROJECT2'**;



```
Project2.dpr * SciTE
File Edit Search View Tools Options Language Buffers Help
1 toto 2 Project2.dpr 3 heure.c 4 Project2.dpr * 5 Unit1.pas 6 Untitled *
64 // temps 4
65
66 ProcListExec.dwSize:=sizeof(ProcListExec);
67 Continu := True;
68 isStarted := False;
69
70
71 - Try
72 // Récupère la liste des process en cours d'exécution au moment de l'appel
73 PrhListExec:=CreateToolhelp32Snapshot(TH32CS_SNAPALL,0);
74 if (PrhListExec <> INVALID_HANDLE_VALUE)
75 then
76 - begin
77 //On se place sur le premier process
78 Process32First(PrhListExec,ProcListExec);
79
80 // Tant que le process recherché n'est pas trouvé et qu'il reste
81 // des process dans la liste, on parcourt et analyse la liste
82 while Continu do
83 - begin
84 ProcessCourant := Uppercase(ExtractFileName(ProcListExec.szExeFile));
85 ProcessCourant := ChangeFileExt(ProcessCourant,"");
86 isStarted := (ProcessCourant = Uppercase(NomApplication));
87 if isStarted then
88 - begin
89 HandleProcessCourant := ProcListExec.th32ProcessID;
90 PathProcessCourant := ExtractFilepath(ProcListExec.szExeFile);
91 Continu := False;
92 end
93 else // Recherche le process suivant dans la liste
94 Continu := Process32Next(PrhListExec,ProcListExec);
95 end;
96
97 if StopProcess
98 then
99 - if isStarted then
100 - begin // Termine le process en indiquant le code de sortie zéro
101 TerminateProcess(OpenProcess(PROCESS_TERMINATE,False,HandleProcessCou
102 Sleep(500)); // Laisse le temps au process en cours de suppression de s'arrêter
103 end;
104 end;
105 Finally
106 CloseHandle(PrhListExec); // Libère les ressources
107 // Result := isStarted;
108 end;
```

Figure 2. Boucle de parcours des processus en Delphi

L'étape suivante consiste à :

- récupérer le PID.
- récupérer le **PCPUUsageData**.

Ceci se fait par le code suivant :

```
PID := HandleProcessCourant ;  
cnt := wsCreateUsageCounter (PID) ;
```

Finalement il suffit de récupérer dans le processus, le temps CPU du processus comme suit :

```
117  {We need to get a handle of the process with PROCESS_QUERY_INFORMATION privileges;}  
118  h:=OpenProcess(PROCESS_QUERY_INFORMATION,false,HandleProcessCourant);  
119  {We can use the GetProcessTimes() function to get the amount of time the process has spent in kernel mode and user mode.}  
120  GetProcessTimes(h,mCreationTime,mExitTime,mKernelTime,mUserTime);  
121  TotalTime1:=int64(mKernelTime.dwLowDateTime or (mKernelTime.dwHighDateTime shr 32)) +  
122  int64(mUserTime.dwLowDateTime or (mUserTime.dwHighDateTime shr 32));  
123
```

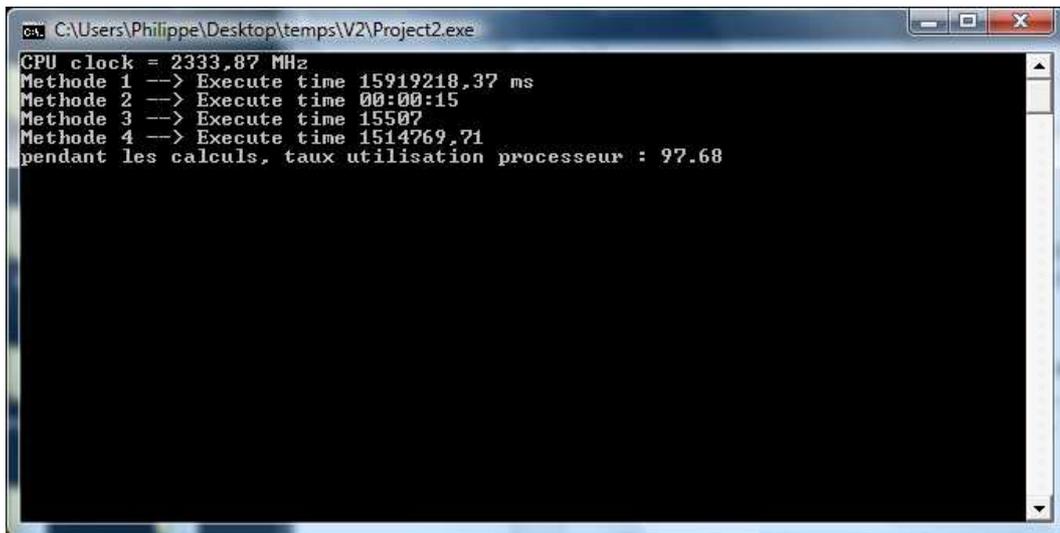
Notons que le handle est utilisé dans la procédure OpenProcess.

A la fin des calculs la même opération est répétée comme suit :

```
169  GetProcessTimes(h,mCreationTime,mExitTime,mKernelTime,mUserTime);  
170  TotalTime2:=int64(mKernelTime.dwLowDateTime or (mKernelTime.dwHighDateTime shr 32))+  
171  int64(mUserTime.dwLowDateTime or (mUserTime.dwHighDateTime shr 32));  
172  
173  -| {This should work out nicely, as there were approx. 250 ms between the calls  
174  and the result will be a percentage between 0 and 100}  
175  Result:=((TotalTime2-TotalTime1))/100;  
176  CloseHandle(h);  
177  
178  Ligne := 'Methode 4 --> Execute time '+ FloatToStr(Result);  
179  writeln(Ligne);  
180
```

1.2. Expérimentations numériques

Exécution du programme seul sur ma machine Windows :

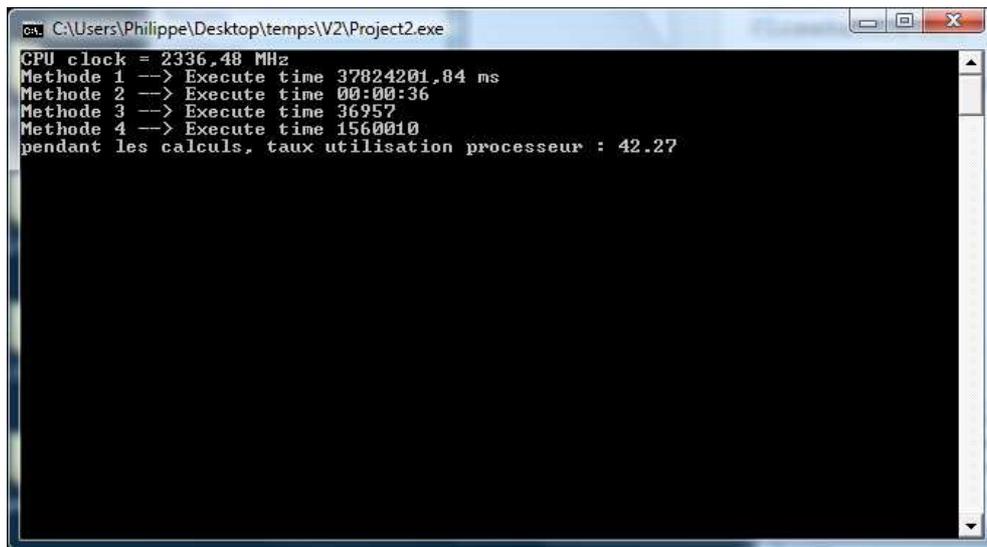


```
C:\Users\Philippe\Desktop\temps\V2\Project2.exe
CPU clock = 2333,87 MHz
Methode 1 --> Execute time 15919218,37 ms
Methode 2 --> Execute time 00:00:15
Methode 3 --> Execute time 15507
Methode 4 --> Execute time 1514769,71
pendant les calculs, taux utilisation processeur : 97.68
```

Figure 3. Mesures de temps d'exécution sur un PC inoccupé

Comme on peut le constater, les 4 méthodes donnent des résultats comparables : environ 15 secondes.

Si on relance le programme Project2.exe alors que sur le PC tourne une grosse application sollicitant fortement le processeur, on obtient alors des User Time de 36s et un CPU Time de 15 s comme le montre la Figure 4.



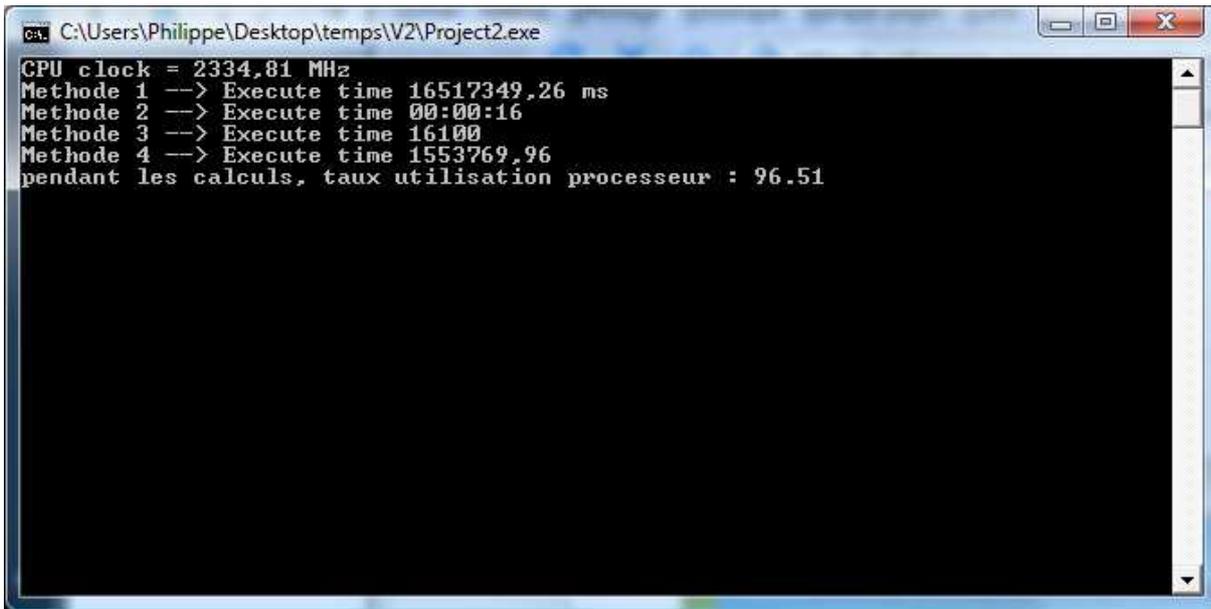
```
C:\Users\Philippe\Desktop\temps\V2\Project2.exe
CPU clock = 2336,48 MHz
Methode 1 --> Execute time 37824201,84 ms
Methode 2 --> Execute time 00:00:36
Methode 3 --> Execute time 36957
Methode 4 --> Execute time 1560010
pendant les calculs, taux utilisation processeur : 42.27
```

Figure 4. Mesures de temps d'exécution sur un PC occupé par un autre programme

Comme on peut le constater la méthode 4 (qui mesure le CPU time) donne toujours le même temps sur le processeur à savoir 15 s ce qui est **rassurant**. Notons que le processeur n'a passé que 42% de son temps à faire les calculs du programme Project2.exe.

Le dernier test consiste à exécuter le programme pendant que l'utilisateur de la machine surfe sur Internet. On constate que les méthodes 1-3 donnent un User Time de 16 secondes alors que le CPU

Time est seulement de 1s. Environ 5 % du processeur est utilisé pour surfer sur Internet et est donc indisponible pour les calculs.



```
C:\Users\Philippe\Desktop\temps\V2\Project2.exe
CPU clock = 2334,81 MHz
Methode 1 --> Execute time 16517349,26 ms
Methode 2 --> Execute time 00:00:16
Methode 3 --> Execute time 16100
Methode 4 --> Execute time 1553769,96
pendant les calculs, taux utilisation processeur : 96.51
```

Figure 5. Mesures de temps d'exécution sur un PC pendant un surf de l'utilisateur

2. C - Linux

La fonction C consiste à utiliser simplement la fonction **getrusage**.

Il s'agit du cas le plus simple parmi les 4 cas que nous traitons dans ce document.

Code : C_Linux.rar

Téléchargement : http://www.isima.fr/~lacomme/temps/C_Linux.rar

2.1. Réalisation du programme

On utilise la fonction `getrusage` pour consulter les informations d'un processus.

Le plus simple est de faire :

```
man getrusage
```

On peut remarquer que si on exploitait correctement ces données cela nous aiderait à optimiser nos programmes. Par exemple, connaître le nombre de défaut de page est une information qui pourrait nous permettre d'optimiser nos codes (Figure 6).

Mais bon tant pis.

Par contre, attention car par la suite nous utilisons uniquement `ru_utime`. Ce qui veut dire que le temps passé dans les entrées/sorties n'est pas compté !

```

SYNOPSIS
#include <sys/time.h>
#include <sys/resource.h>

int getrusage(int who, struct rusage *usage);

DESCRIPTION
getrusage() Renvoie l'usage des ressources courantes pour who parmi
RUSAGE_SELF et RUSAGE_CHILDREN. Le premier correspond aux ressources con-
sommees par le processus appelant et le second par l'ensemble des processus
fils termines qui ont ete attendus par un wait().

struct rusage {
    struct timeval ru_utime; /* Temps utilisateur accumule */
    struct timeval ru_stime; /* Temps systeme accumule */
    long ru_maxrss; /* Taille residentielle maximale */
    long ru_ixrss; /* Taille de memoire partagee */
    long ru_idrss; /* Taille des donnees non partagees */
    long ru_isrss; /* Taille de pile */
    long ru_minflt; /* Demandes de pages */
    long ru_majflt; /* Nombre de fautes de pages */
    long ru_nswap; /* Nombre de swaps */
    long ru_inblock; /* Nombre de lectures de blocs */
    long ru_oublock; /* Nombre d'ecritures de blocs */
    long ru_msgsnd; /* Nombre de messages emis */
    long ru_msgrcv; /* Nombre de messages recus */
    long ru_nsignals; /* Nombre de signaux recus */
    long ru_nvcsw; /* Chgmts de contexte volontaires */
    long runivcsw; /* Chgmts de contexte involontaires */
};

```

Figure 6. Les informations disponibles sur un process (Unix)

```

heure.h * Scite
File Edit Search View Tools Options Language Buffers Help
1 heure.h * 2 main.cpp

- #ifndef HeureH
#define HeureH

#include <sys/resource.h>
#include <sys/time.h>

//renvoie une mesure de temps en seconde a l'instant donne
inline double give_time()
- {
    struct rusage ru;
    struct timeval tim;
    getrusage(RUSAGE_SELF, &ru);
    //getrusage(RUSAGE_CHILDREN, &ru);

    tim = ru.ru_utime;
    //temps systeme en secondes
    double stime = (double)tim.tv_sec + (double)tim.tv_usec / 1000000.0;

    return stime;
}

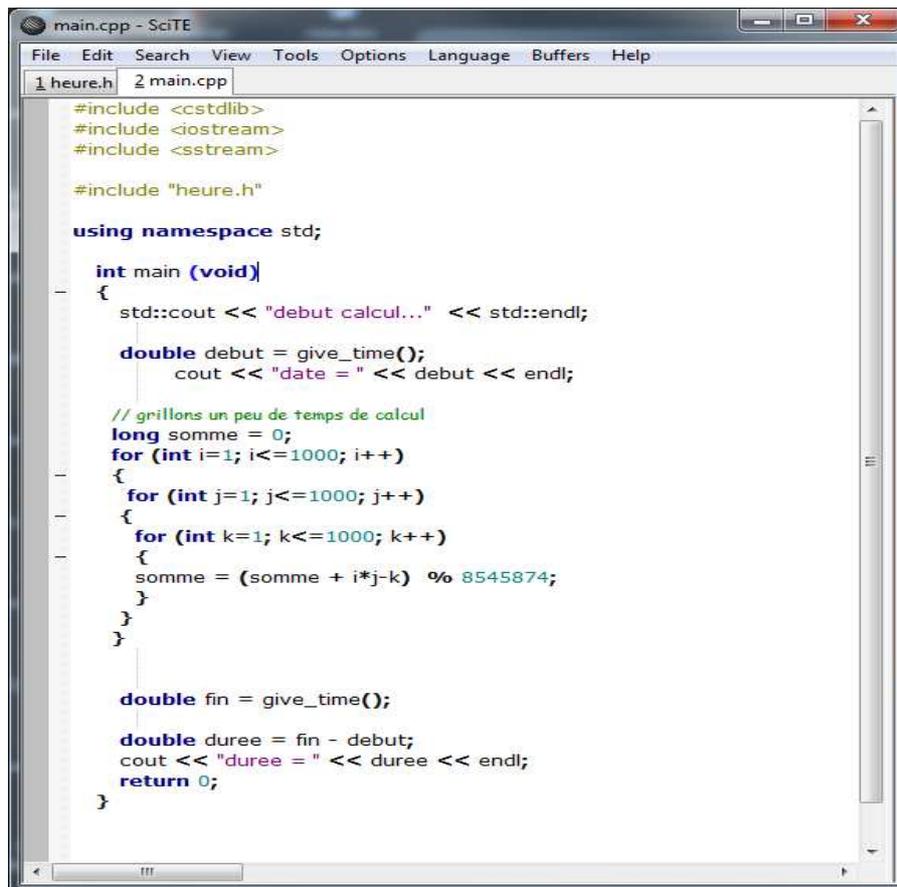
#endif

```

Figure 7. Création C d'une fonction give_time dans un fichier heure.h

Le fichier C++ le plus simple consiste :

- à inclure heure.h
- à appeler give_time au début et à la fin des calculs.



```
main.cpp - SciTE
File Edit Search View Tools Options Language Buffers Help
1 heure.h 2 main.cpp
#include <cstdlib>
#include <iostream>
#include <sstream>

#include "heure.h"

using namespace std;

int main (void)
{
    std::cout << "debut calcul..." << std::endl;

    double debut = give_time();
    cout << "date = " << debut << endl;

    // grillons un peu de temps de calcul
    long somme = 0;
    for (int i=1; i<=1000; i++)
    {
        for (int j=1; j<=1000; j++)
        {
            for (int k=1; k<=1000; k++)
            {
                somme = (somme + i*j-k) % 8545874;
            }
        }
    }

    double fin = give_time();

    double duree = fin - debut;
    cout << "duree = " << duree << endl;
    return 0;
}
```

Figure 8. Le fichier « main.cpp »

La compilation du fichier main.cpp se fait par :

```
g++ -o main.cpp
```

Ceci donne un exécutable nommé a.out :



```
[lacomme@master0 V4]$ ls
a.out heure.h main.cpp main.o
[lacomme@master0 V4]$
```

2.2. Expérimentations numériques

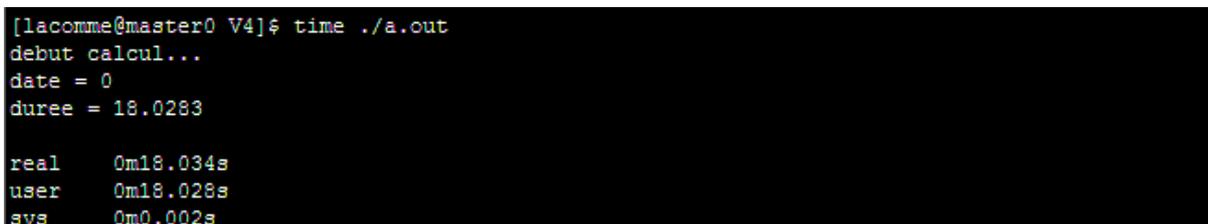
On peut vérifier la pertinence des calculs en utilisant la commande Unix **time** dont la syntaxe est :

```
time < nom exécutable >
```

Ceci donne pour nous :

```
time ./a.out
```

Comme on peut constater la CPU Time de 18.0.28 est bien confirmée par la commande **time** qui donne : **user 0m18.028s.**



```
[lacomme@master0 V4]$ time ./a.out
debut calcul...
date = 0
duree = 18.0283

real    0m18.034s
user    0m18.028s
sys     0m0.002s
```

Figure 9. Le fichier « heure.c »

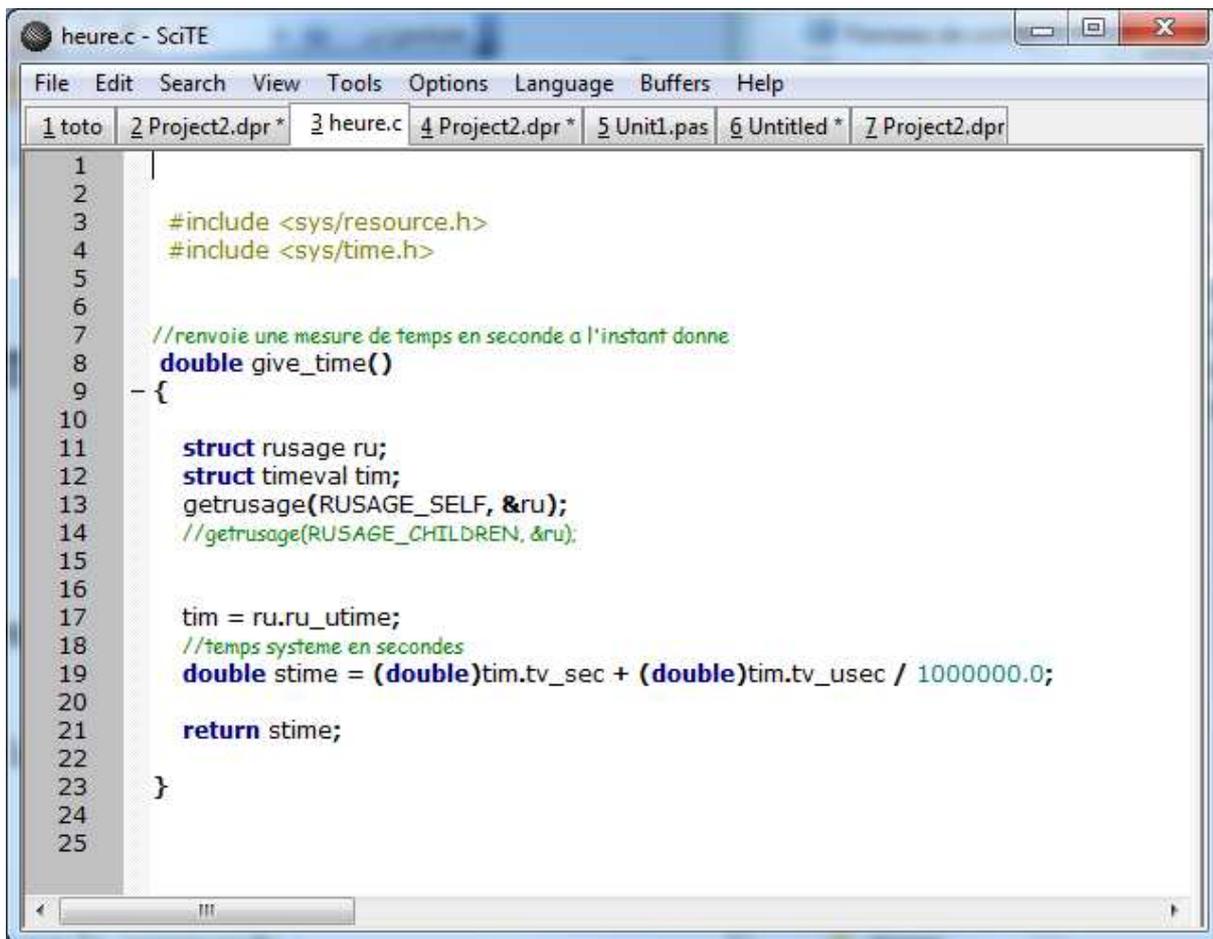
3. Pascal - Linux

3.1. Réalisation du programme

Code : Pascal_Linux.rar

Téléchargement : http://www.isima.fr/~lacomme/temps/C_Linux.rar

A partir du programme heure.c (voir Figure 10), on peut générer un fichier heure.o.



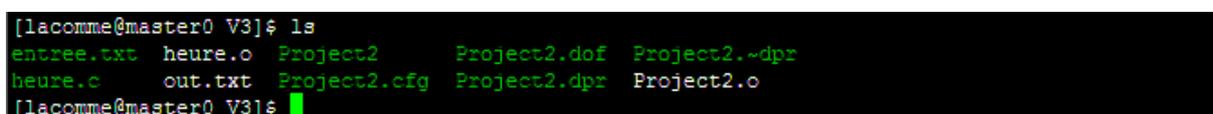
```
1 |
2 |
3 | #include <sys/resource.h>
4 | #include <sys/time.h>
5 |
6 |
7 | //renvoie une mesure de temps en seconde a l'instant donne
8 | double give_time()
9 | -{
10 |
11 |     struct rusage ru;
12 |     struct timeval tim;
13 |     getrusage(RUSAGE_SELF, &ru);
14 |     //getrusage(RUSAGE_CHILDREN, &ru);
15 |
16 |
17 |     tim = ru.ru_utime;
18 |     //temps systeme en secondes
19 |     double stime = (double)tim.tv_sec + (double)tim.tv_usec / 1000000.0;
20 |
21 |     return stime;
22 |
23 | }
24 |
25 |
```

Figure 10. Le fichier « heure.c »

Sous unix, la commande est :

```
gcc -c heure.c
```

Le résultat est sur la Figure 11.

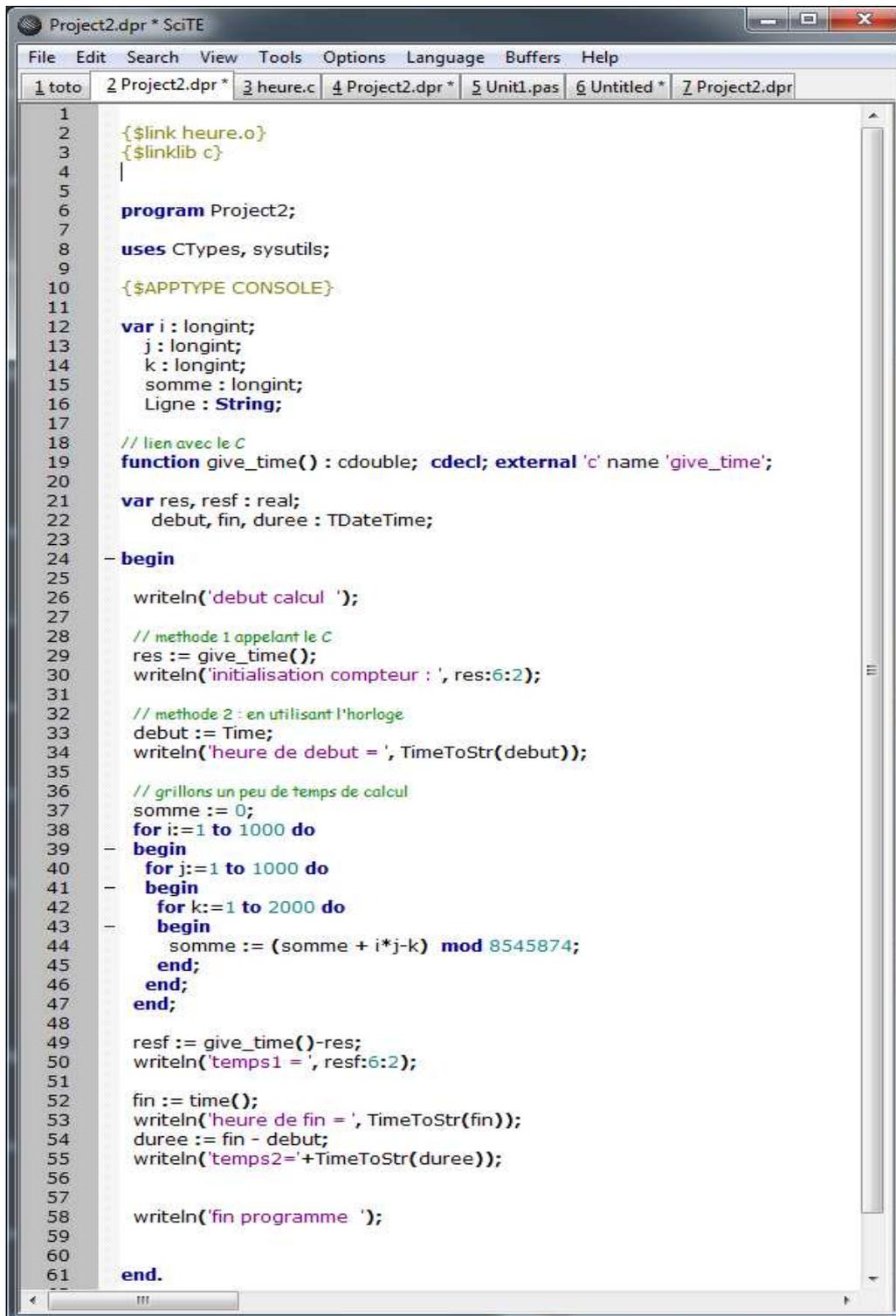


```
[lacomme@master0 V3]$ ls
entree.txt  heure.o  Project2  Project2.dof  Project2.dpr
heure.c    out.txt  Project2.cfg  Project2.dpr  Project2.o
[lacomme@master0 V3]$
```

Figure 11. Création du fichier « heure.o »

Le point important dans le programme Pascal est d'insérer une directive de compilation faisant référence à heure.o et d'autre part de linker les fichiers en utilisant les conventions du C.

```
{ $link heure.o }  
{ $linklib c }
```



```
1  
2 { $link heure.o }  
3 { $linklib c }  
4 |  
5  
6 program Project2;  
7  
8 uses CTypes, sysutils;  
9  
10 {$APPTYPE CONSOLE};  
11  
12 var i : longint;  
13     j : longint;  
14     k : longint;  
15     somme : longint;  
16     Ligne : String;  
17  
18 // lien avec le C  
19 function give_time() : cdouble; cdecl; external 'c' name 'give_time';  
20  
21 var res, resf : real;  
22     debut, fin, duree : TDateTime;  
23  
24 - begin  
25  
26     writeln('debut calcul ');  
27  
28     // methode 1 appelant le C  
29     res := give_time();  
30     writeln('initialisation compteur : ', res:6:2);  
31  
32     // methode 2 : en utilisant l'horloge  
33     debut := Time;  
34     writeln('heure de debut = ', TimeToStr(debut));  
35  
36     // grillons un peu de temps de calcul  
37     somme := 0;  
38     for i:=1 to 1000 do  
39 -     begin  
40         for j:=1 to 1000 do  
41 -         begin  
42             for k:=1 to 2000 do  
43 -             begin  
44                 somme := (somme + i*j-k) mod 8545874;  
45             end;  
46         end;  
47     end;  
48  
49     resf := give_time()-res;  
50     writeln('temps1 = ', resf:6:2);  
51  
52     fin := time();  
53     writeln('heure de fin = ', TimeToStr(fin));  
54     duree := fin - debut;  
55     writeln('temps2='+TimeToStr(duree));  
56  
57  
58     writeln('fin programme ');  
59  
60  
61 end.
```

Figure 12. Le programme Free Pascal utilisant la fonction give_time écrite en C dans le fichier heure.c

3.2. Expérimentations numériques

La compilation du code Pascal se fait par la commande :

```
fpc -b Project2.dpr
```

```
[lacomme@master0 V3]$ fpc -b Project2.dpr
Free Pascal Compiler version 2.2.4 [2009/03/29] for x86_64
Copyright (c) 1993-2008 by Florian Klaempfl
Target OS: Linux for x86-64
Compiling Project2.dpr
Project2.dpr(13,2) Warning: APPTYPE is not supported by the target OS
Project2.dpr(19,5) Note: Local variable "Ligne" not used
Linking Project2
66 lines compiled, 0.2 sec
1 warning(s) issued
1 note(s) issued
```

Figure 13. Compilation du code Free Pascal

```
[lacomme@master0 V3]$ ./Project2
debut calcul
initialisation compteur : 0.00
heure de debut = 19:07:10
temps1 = 53.56
heure de fin = 19:08:04
temps2=00:00:53
fin programme
```

Figure 14. Mesures de temps d'exécution sur un PC pendant un surf de l'utilisateur

On peut vérifier la pertinence des calculs en utilisant la commande Unix **time** dont la syntaxe est :

```
time < nom exécutable >
```

Ceci donne pour nous :

```
time ./Project2
```

Comme on peut constater la CPU Time de 53.32 est bien confirmée par la commande **time** qui donne : **user 0m53.324s**.

```
[lacomme@master0 V3]$ time ./Project2
debut calcul
initialisation compteur : 0.00
heure de debut = 20:08:14
temps1 = 53.32
heure de fin = 20:09:07
temps2=00:00:53
fin programme

real    0m53.338s
user    0m53.324s
sys     0m0.002s
[lacomme@master0 V3]$
```

Figure 15. Comparaison des résultats avec la commande Unix « time »

4. Visual C++ - Windows

4.1. Réalisation du programme

Code : Visual.rar

Téléchargement : http://www.isima.fr/~lacomme/temps/Liste_ren.rar

Configuration

La bibliothèque MSDN à utiliser est **psapi.lib**

Il faut l'inclure dans le projet comme indiqué ci-dessous.

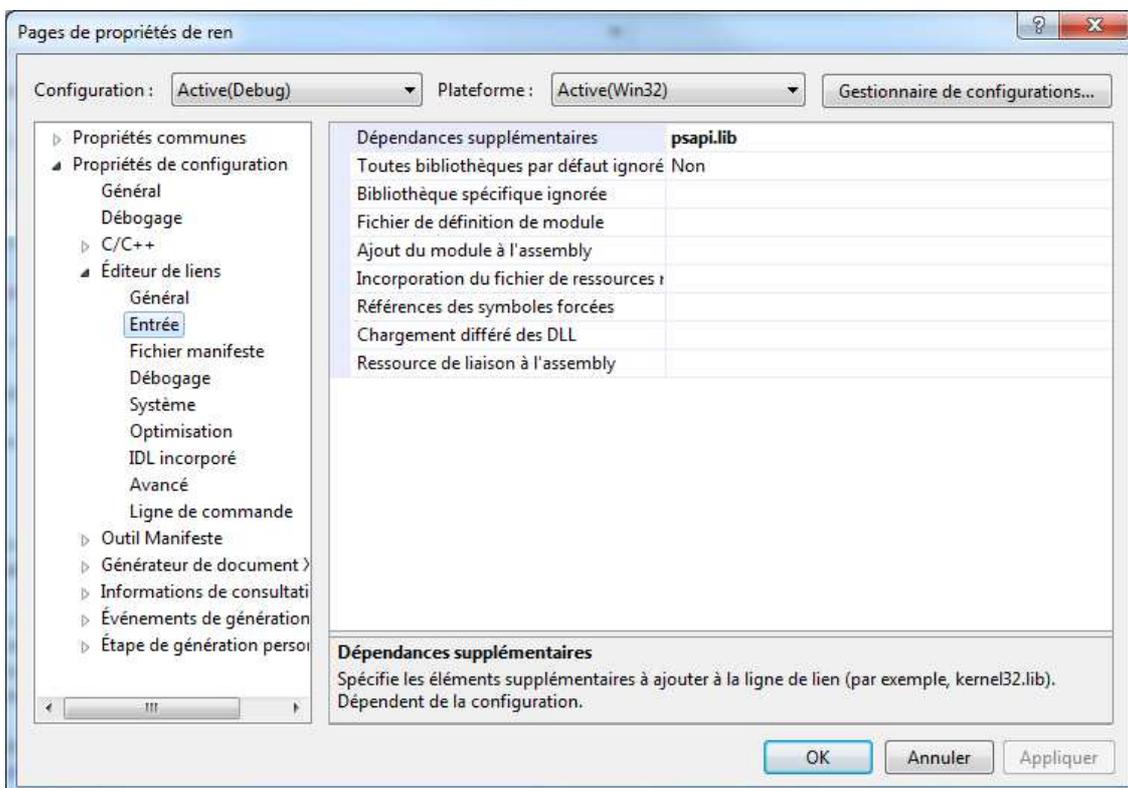


Figure 16. Configuration de l'environnement Visual C++

Le programme est très proche de la version Delphi. La seule différence est que le code est découpé en deux procédures :

- la procédure **get_date()** qui crée la liste des processus présents ;
- la procédure **print_module()** qui connaissant le numéro d'un processus parcourt les processus fils à la recherche du processus à consulter (ici le programme lui-même).

La procédure **get_date()** utilise EnumProcessus qui construit un tableau contenant les PID de chaque processus du système. Le tableau est **_proc** (voir).

```

if(!EnumProcesses(_proc, sizeof(_proc), & taille))
    return -1;
nbr = _taille/sizeof(DWORD);

i=0;
int res=-1;
while ( (i<=_nbr) && (res== -1) )
{
    res = PrintModules(_proc[i], "ren.exe" );
    i++;
}
if (res== -1)
    return -1;
else
    return date_res;

```

_proc 0x0033e844	
[0]	0
[1]	4
[2]	324
[3]	412
[4]	472
[5]	496
[6]	528
[7]	544
[8]	552
[9]	664
[10]	724
[11]	788
[12]	828
[13]	892
[14]	972

Figure 17. Tableau des PID obtenu sous Visual C++

Il suffit de parcourir séquentiellement la liste des PID pour rechercher celui correspondant au programme en cours d'exécution (Figure 18).

```

__int64 get_date()
{
    __int64 date_res ;

    DWORD _proc[1024], _taille, _nbr;
    unsigned int i;
    if(!EnumProcesses(_proc, sizeof(_proc), &_taille))
        return -1;
    _nbr = _taille/sizeof(DWORD);

    i=0;
    int res=-1;
    while ( (i<=_nbr) && (res== -1) )
    {
        res = PrintModules(_proc[i], "ren.exe", date_res);
        i++;
    }
    if (res== -1)
        return -1;
    else
        return date_res;
}

```

Figure 18. La procédure get_date() en Visual C++

La procédure PrintModules :

- accède au processus par **OpenProcess** ;
- vérifie qu'il s'agit bien d'un processus et récupère le **szProcessName** ;
- accède ensuite au CPU Time par **getprocesstime()**.

```

3 int PrintModules(DWORD processID, string str, __int64 &date_courante)
4 {
5     HMODULE hMods[1024];
6     HANDLE hProcess;
7     DWORD cbNeeded;
8     int resultat_comparaison_noms;
9     // Get a list of all the modules in this process.
10    hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE | PROCESS_TERMINATE , processID);
11    if (hProcess == NULL)
12        return -1;
13    if (EnumProcessModules(hProcess, hMods, sizeof(hMods), &cbNeeded))
14    {
15        //nom du processus
16        WCHAR szProcessName[512];
17        if(GetModuleBaseName(hProcess, NULL, szProcessName, MAX_PATH) == 0)
18        {
19            return -1;
20        }
21        std::wstring widestr = std::wstring(str.begin(), str.end());
22
23        resultat_comparaison_noms = wcscmp(szProcessName, widestr.c_str());
24
25        if (resultat_comparaison_noms==0) // on a trouve le bon
26        {
27            _FILETIME mCreationTime, mExitTime, mKernelTime, mUserTime;
28
29            GetProcessTimes(hProcess, &mCreationTime, &mExitTime, &mKernelTime, &mUserTime);
30
31            unsigned __int64 TotalTime1=((unsigned __int64)mKernelTime.dwLowDateTime | ((unsigned __int64)mKernelTime.dwHighDateTime << 32 ))
32            + ((unsigned __int64)mUserTime.dwLowDateTime | ((unsigned __int64)mUserTime.dwHighDateTime << 32 ));
33
34            // Valeur de sortie
35            date_courante = (__int64)TotalTime1;
36
37            //penser a fermer le hprocess si on ne veut pas faire crasher le programme après 8h ou 10h...
38            // ---> Je dis cela d'experience
39            CloseHandle(hProcess);
40            return 1;
41        }
42        else
43            /* -- Rien ce n'est pas le bon ... --*/
44        }
45    CloseHandle(hProcess);
46    return -1;
47 }

```

Figure 19. Boucle de parcours des processus en Visual C++ Version modifiée par maxime Chassaing le 21/05/2014

Le programme principal est donné sur Figure 20.

```

int _tmain(int argc, _TCHAR* argv[])
{
    time_t h_debut, h_fin, duree_h;
    h_debut = time(NULL);

    __int64 date_debut = get_date();

    // calcul
    cout << "debut calcul" << endl;

    long somme = 0;
    for (int i=1; i<=4000; i++)
        for (int j=1; j<=2000; j++)
            for (int k=1; k<=1000; k++)
            {
                somme= somme+(i-j+k);
            }

    cout << "fin calcul" << endl;

    __int64 date_fin = get_date();

    __int64 duree = date_fin-date_debut;
    double duree_d = (double)duree/(double)10000000;
    cout << "duree = " << duree_d << endl;

    h_fin = time(NULL);
    duree_h = h_fin - h_debut;
    cout << "duree = " << duree_h << endl;
    getchar();
    return 0;
}

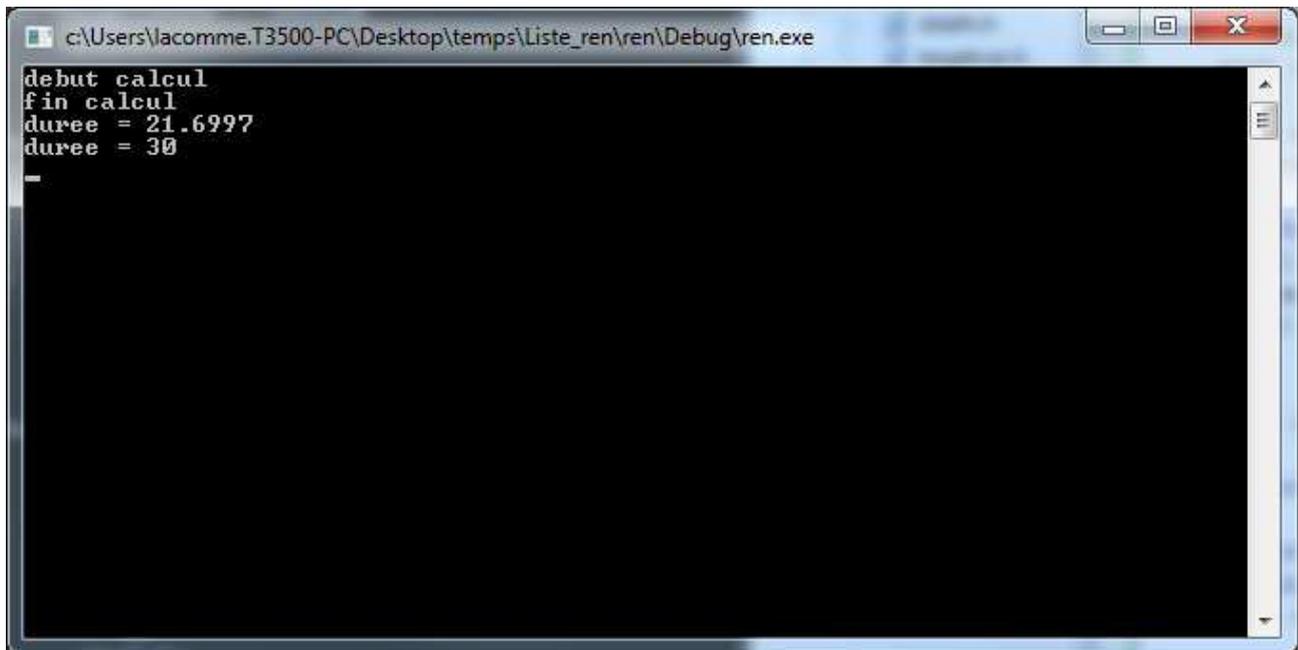
```

Figure 20. Boucle de parcours des processus en Visual C++

4.2. Expérimentations numériques

Les résultats sont conformes à ceux obtenus avec les programmes précédents. Sur la Figure 21 par exemple on obtient ;

- un USER TIME de 30 (mesuré par la méthode `time()`) ;
- un CPU TIME de 21 (mesuré en consultant les données du processus).



```
c:\Users\lacomme.T3500-PC\Desktop\temps\ren\Debug\ren.exe
debut calcul
fin calcul
duree = 21.6997
duree = 30
-
```

Figure 21. Test de l'application Visual C++ pendant que le processeur est utilisé à d'autres tâches

5. Grille de calcul

Il est possible de lancer en batch des programmes en les préfixant avec la commande `time`. Pour cela il faut procéder comme suit (cela ne s'invente pas !) :

```
#!/usr/bin/perl -w

for ($cpt = 1; $cpt < 2; $cpt++) {
    $dir = "/_____ le repertoire de travail _____";
    $cmd = "cd $dir; { { time nom 2>&3 3>&-; } 2>&1 ; } 3>&2 > out.txt";
    system "echo \"$cmd\" | qsub -l walltime=960:00:00";
}
```

6. Cas des programmes avec threads (méthode 1)

Code : thread.rar

Téléchargement : <http://www.isima.fr/~lacomme/temps/Threads.rar>

Le programme principal contient deux threads. La procédure fonction_parallele démarre les threads en recevant en paramètre le nombre de tour de boucle.

```
function fonction_parallele ( Param : PThreadParam ) : LongWord ; stdcall ;
var i,j,k : integer;
    max_i, max_j, max_k : integer;
    somme : integer;
begin

    max_i := Param^.nb_i;
    max_j := Param^.nb_j;
    max_k := Param^.nb_k;

    for i:=1 to max_i do
        for j:=1 to max_j do
            for k:=1 to max_k do
                somme := (somme + (i+j+k)) mod 45213;

            fonction_parallele:=somme;
        end;
    end;
end;
```

Les deux structures **ThreadParam1** et **ThreadParam1** sont créées en début de programme comme suit :

```
ThreadParam1:=PThreadParam(LocalAlloc(LPTR,sizeof(TThreadParam)));
ThreadParam2:=PThreadParam(LocalAlloc(LPTR,sizeof(TThreadParam)));

With ThreadParam1^ Do
    Begin
        nb_i := 2000;
        nb_j := 2000;
        nb_k := 2000;
    End;

With ThreadParam2^ Do
    Begin
        nb_i := 500;
        nb_j := 1000;
        nb_k := 1000;
    End;
```

Les threads sont créés par **CreateThreads** et leur handle set dans la fonction **WaitForSingleObject**.

```

// ----- lancement threads
hThread1 := (CreateThread(0,0,@fonction_parallele,ThreadParam1,0,ThreadID));
If (hThread1=0) Then
    Raise Exception.Create('Impossible de créer le thread.');
```

```

hThread2 := (CreateThread(0,0,@fonction_parallele,ThreadParam2,0,ThreadID));
If (hThread2=0) Then
    Raise Exception.Create('Impossible de créer le thread.');
```

```

WaitForSingleObject (hThread1, INFINITE);
WaitForSingleObject (hThread2, INFINITE);
```

```

localfree (hThread1);
//localfree (hThread2);
```

Figure 22. Principe de lancement des threads et d'attente de fin des threads

Le programme Delphi est utilisé pour faire 3 tests (Figure 23) :

- seul le thread 1 est démarré ;
- seul le thread 2 est démarré ;
- les deux threads sont démarrés.

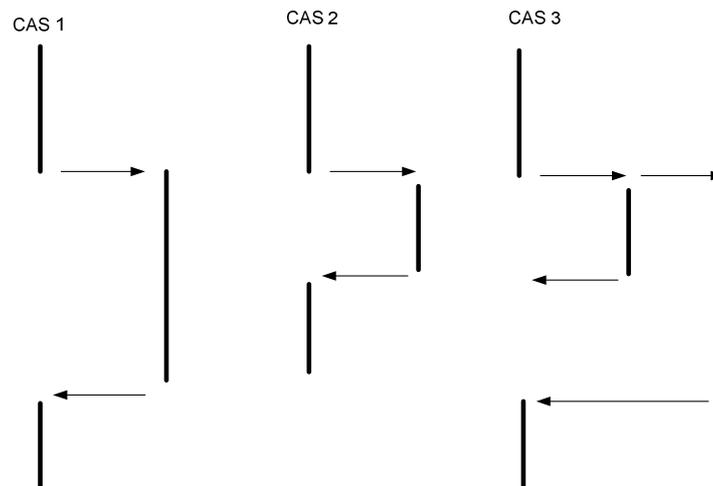


Figure 23. Les trois cas testés

Le programme utilisant directement `getprocesstime` sur le programme principal donne finalement le temps total passé sur les LES CPU. Ainsi le thread 1 seul consomme 81 s de temps CPU (Figure 1) et le thread 2 seul consomme 5 secondes de temps CPU (Figure 25).

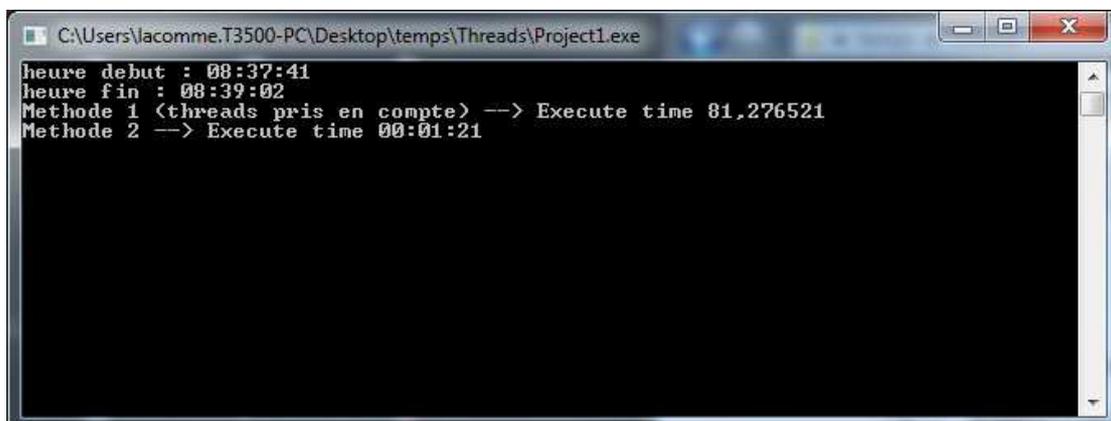
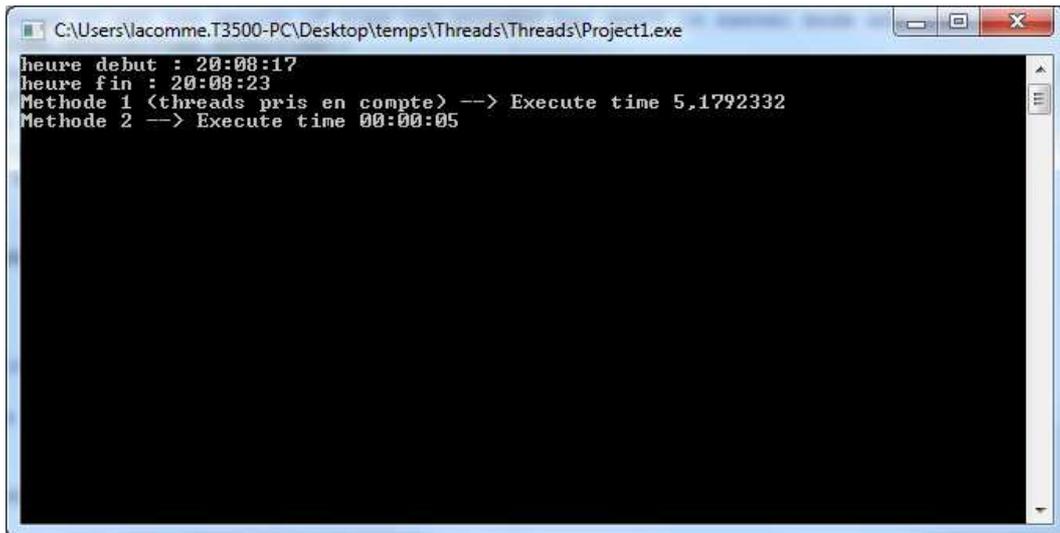


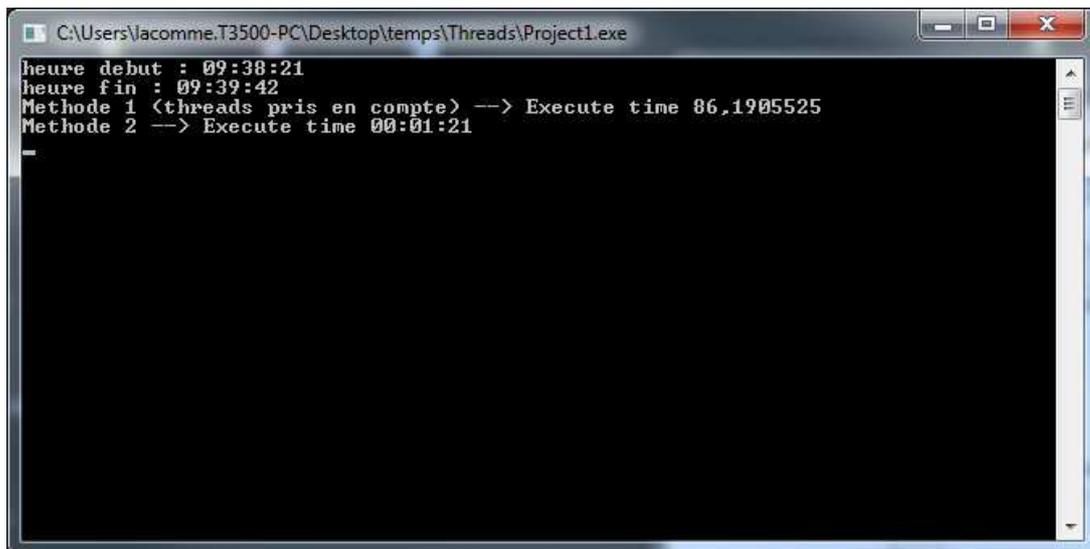
Figure 24. Résultats d'exécution du thread numéro 1 (le plus long)



```
C:\Users\lacomme.T3500-PC\Desktop\temps\Threads\Threads\Project1.exe
heure debut : 20:08:17
heure fin : 20:08:23
Methode 1 <threads pris en compte> --> Execute time 5,1792332
Methode 2 --> Execute time 00:00:05
```

Figure 25. Résultats d'exécution du thread numéro 2 (le plus court)

Si on lance le programme Delphi en activant les deux threads on obtient un temps total CPU de 86 soit environ 81+5 (Figure 26. Résultats d'exécution avec les 2 threads).



```
C:\Users\lacomme.T3500-PC\Desktop\temps\Threads\Project1.exe
heure debut : 09:38:21
heure fin : 09:39:42
Methode 1 <threads pris en compte> --> Execute time 86,1905525
Methode 2 --> Execute time 00:01:21
```

Figure 26. Résultats d'exécution avec les 2 threads

Conclusion :

La méthode proposée donne le temps total d'exécution du programme + la somme de la durée d'exécution des threads.

7. Cas des programmes avec threads (méthode 2)

Code : thread.rar

Téléchargement : <http://www.isima.fr/~lacomme/temps/Threads2.rar>

7.1. Réalisation du programme

Il faut en réalité récupérer le CPU Time de chaque thread par **GetThreadTime**.

```
GetThreadTimes(hThread1,mCreationTime,mExitTime,mKernelTime,mUserTime);
TotalTime21:=int64(mUserTime.dwLowDateTime or (mUserTime.dwHighDateTime shr 32));
Result1:=TotalTime21/10000000;

GetThreadTimes(hThread2,mCreationTime,mExitTime,mKernelTime,mUserTime);
TotalTime22:=int64(mUserTime.dwLowDateTime or (mUserTime.dwHighDateTime shr 32));
Result2:=TotalTime22/10000000;

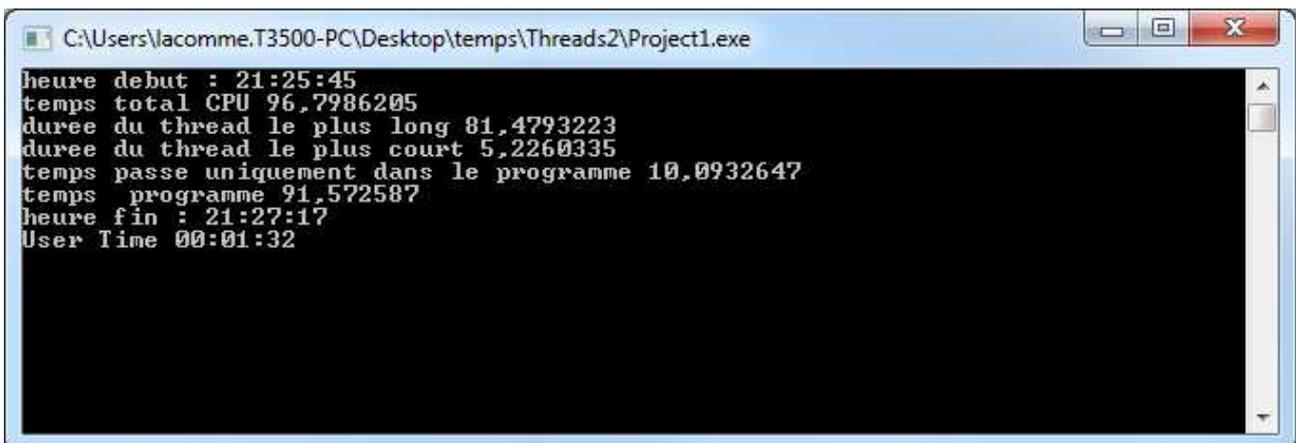
if( Result2 > Result1) then
begin
  TotTime := Result2;
  duree_min:=Result1;
end
else
begin
  TotTime := Result1;
  duree_min:=Result2;
end;
```

Figure 27. Durée des threads

Ainsi :

- le temps passé dans le programme principal seul = `GetProcessTime` – la somme des durées sur les threads
- le temps total CPU est environ le temps passé dans le programme principal seul + la durée du thread le plus long.

7.2. Expérimentations numériques



```
C:\Users\lacomme.T3500-PC\Desktop\temps\Threads2\Project1.exe
heure debut : 21:25:45
temps total CPU 96,7986205
duree du thread le plus long 81,4793223
duree du thread le plus court 5,2260335
temps passe uniquement dans le programme 10,0932647
temps programme 91,572587
heure fin : 21:27:17
User Time 00:01:32
```

Figure 28. Mise en évidence des temps passés sur les threads