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 utiliées :

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 méhode 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 :

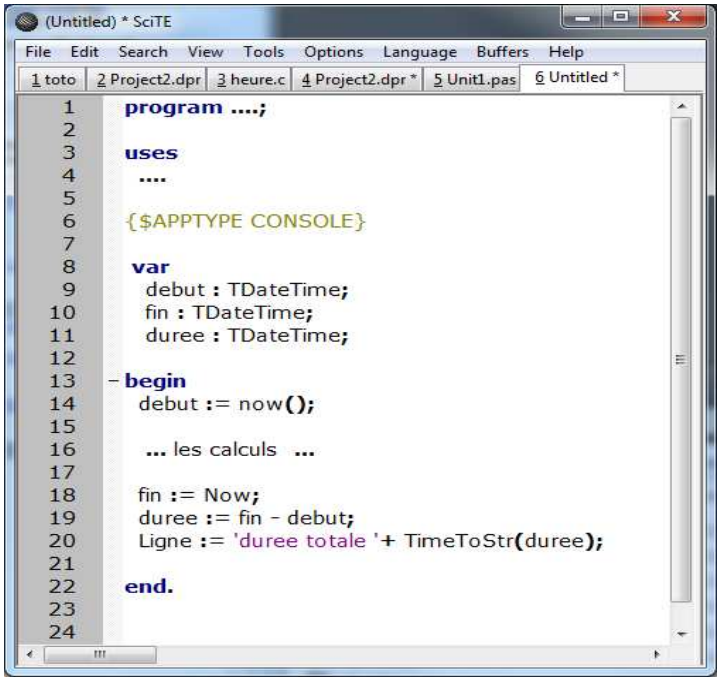

Figure 1. Programme Delphi usuel

Cette manière de procéder conduit à mesure 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 ;
- effecter 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';**

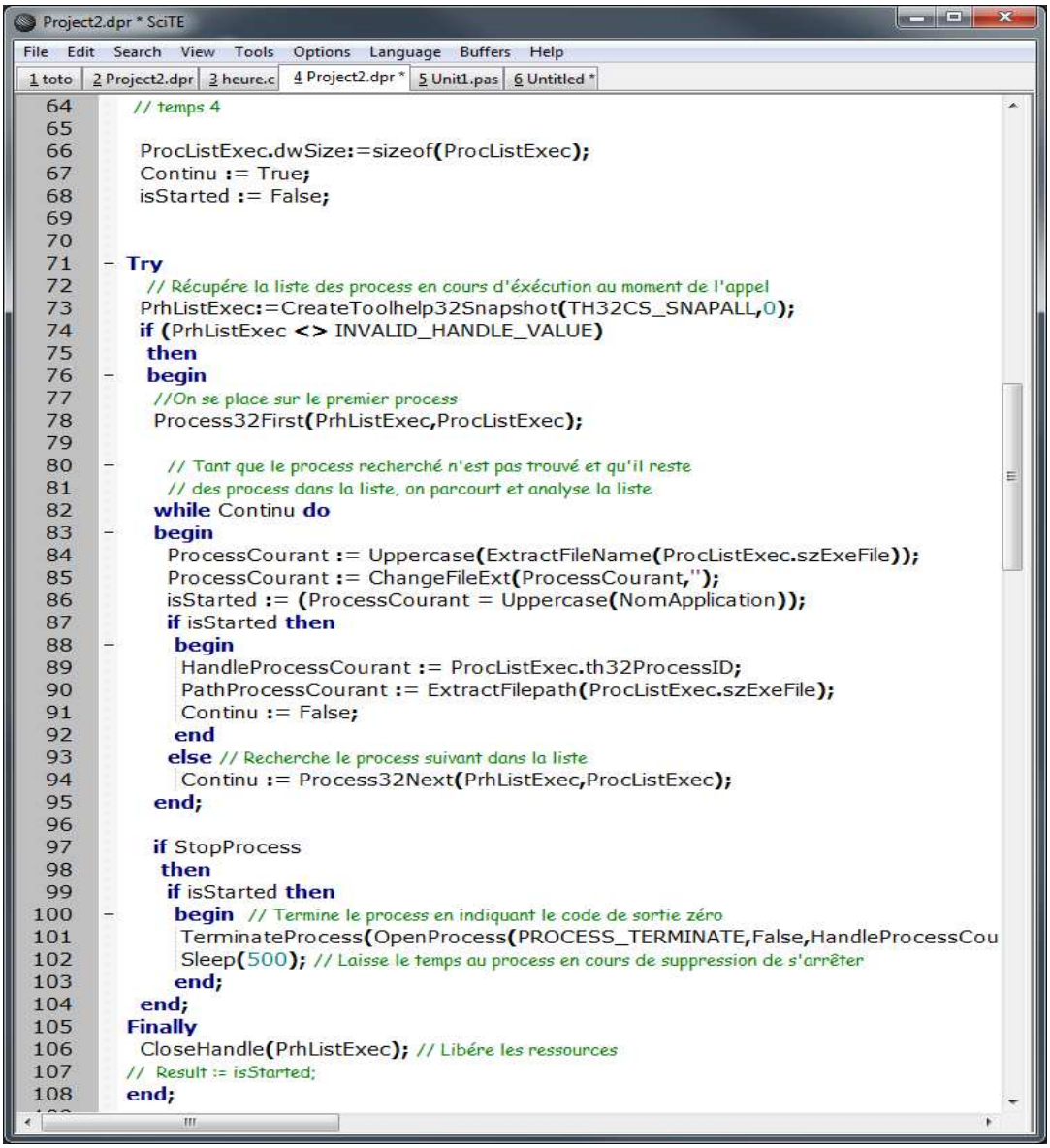

Figure 2. Boucle de parcours des processus en Delphi

L’étape suivant consiste à :

- récupérer le PID.

- récupérer le **PCPUUsageData.**

Ceci se fait par le code suivant :

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

Finallement il suffit de récuperer 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 procedure 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 :

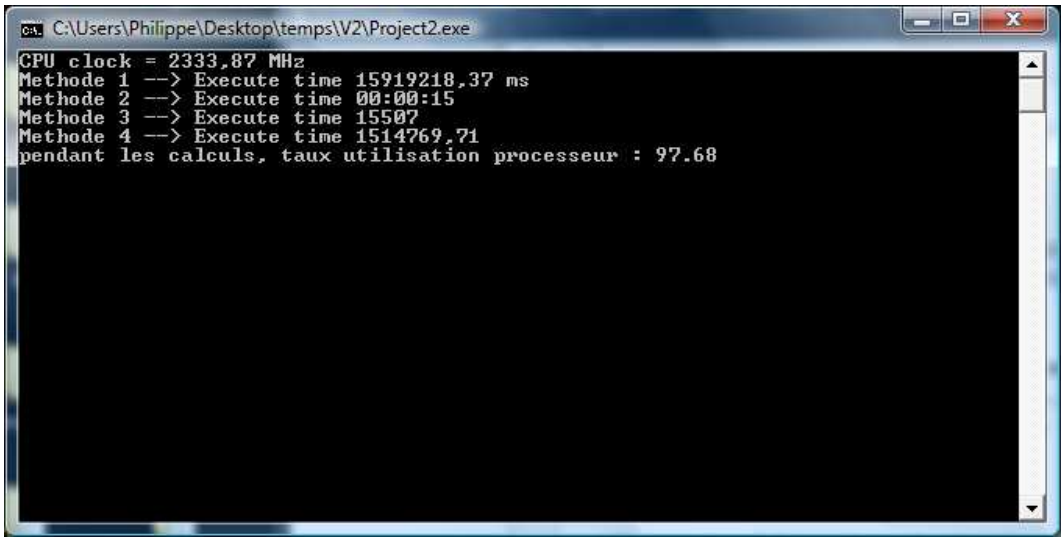

Figure 3. Mesures de temps d'exécution sur un PC inoccupé

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.

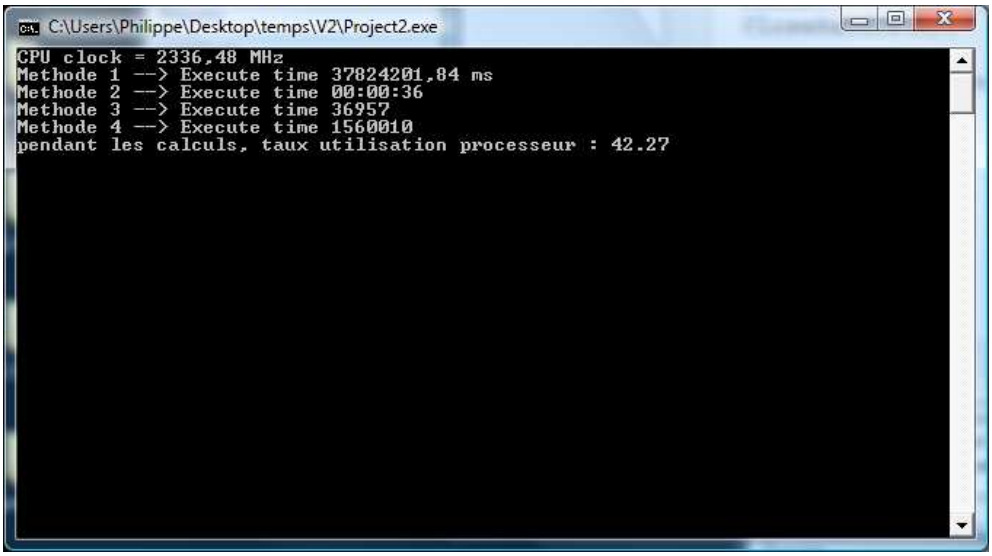

Figure 4. Mesures de temps d'exécution sur un PC occupé par un autre programme

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. Envison 5 % du processeur est utilisé pour surfer sur Internet et est donc indisponible pour les calculs.

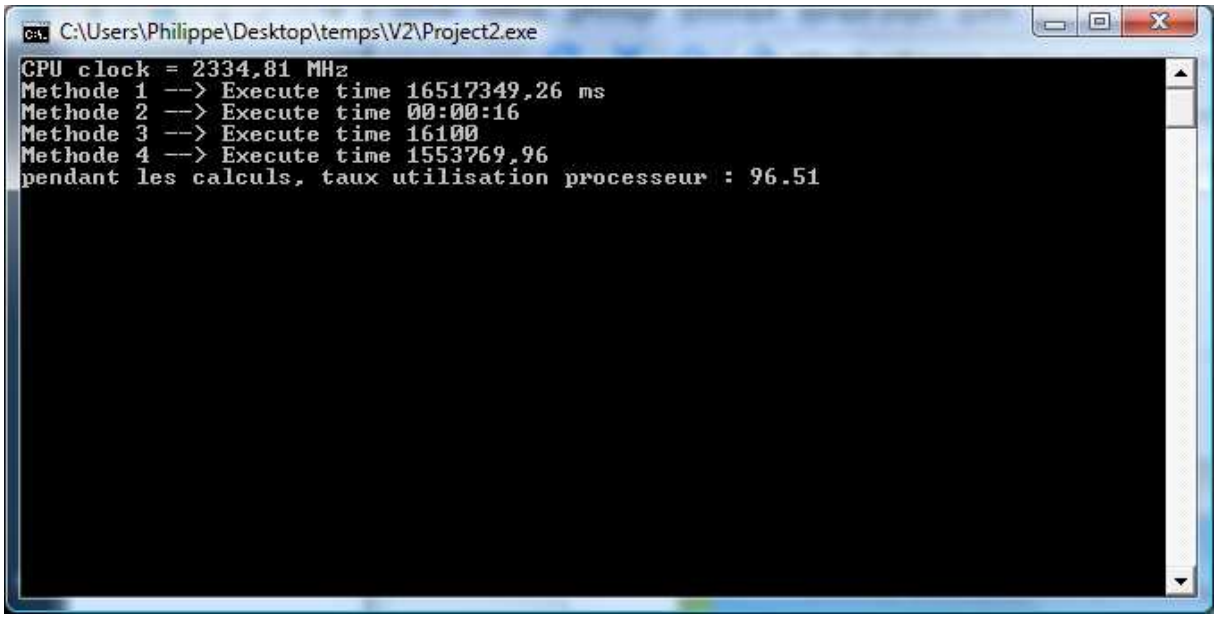

Figure 5. Mesures de temps d'exécution sur un PC pendant un surf de l'utilisateur

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 processu.

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, connaitre 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  lâ€™utilisation  des  ressources  courantes pour who parmi
       RUSAGE SELF et RUSAGE CHILDREN.  Le premier  correspond  aux  ressources  con-
       sommÃ©es  par  le  processus appelant et le second par lâ€™ensemble des processus
       fils terminÃ©s qui ont Ã©tÃ© attendus par un wait().

            struct rusage {
              struct timeval ru_utime; /* Temps utilisateur Ã©coulÃ©        */
              struct timeval ru_stime; /* Temps systÃ¨me Ã©coulÃ©            */
              long   ru_maxrss;        /* Taille rÃ©sidente maximale       */
              long   ru_ixrss;         /* Taille de mÃ©moire partagÃ©e      */
              long   ru_idrss;         /* Taille des donnÃ©es non partagÃ©es */
              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 dâ€™Ã©critures de blocs     */
              long   ru_msgsnd;        /* Nombre de messages Ã©mis         */
              long   ru_msgrcv;        /* Nombre de messages reÃ§us        */
              long   ru_nsignals;      /* Nombre de signaux reÃ§us         */
              long   ru_nvcsw;         /* Chgmnts de contexte volontaires */
              long   ru_nivcsw;        /* Chgmnts de contexte involontaires*/
            };
:
```

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

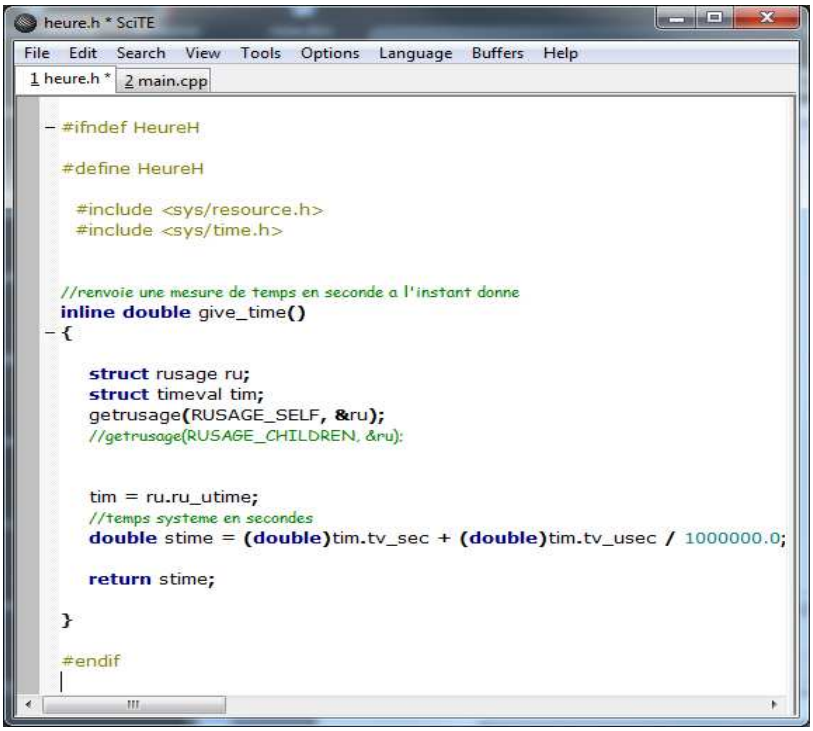

Figure 7. Création C d'une foncton give_time dans un fichier heure.h

Le fichier C++ le plus simple consiste :

- à inclure heure.h

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

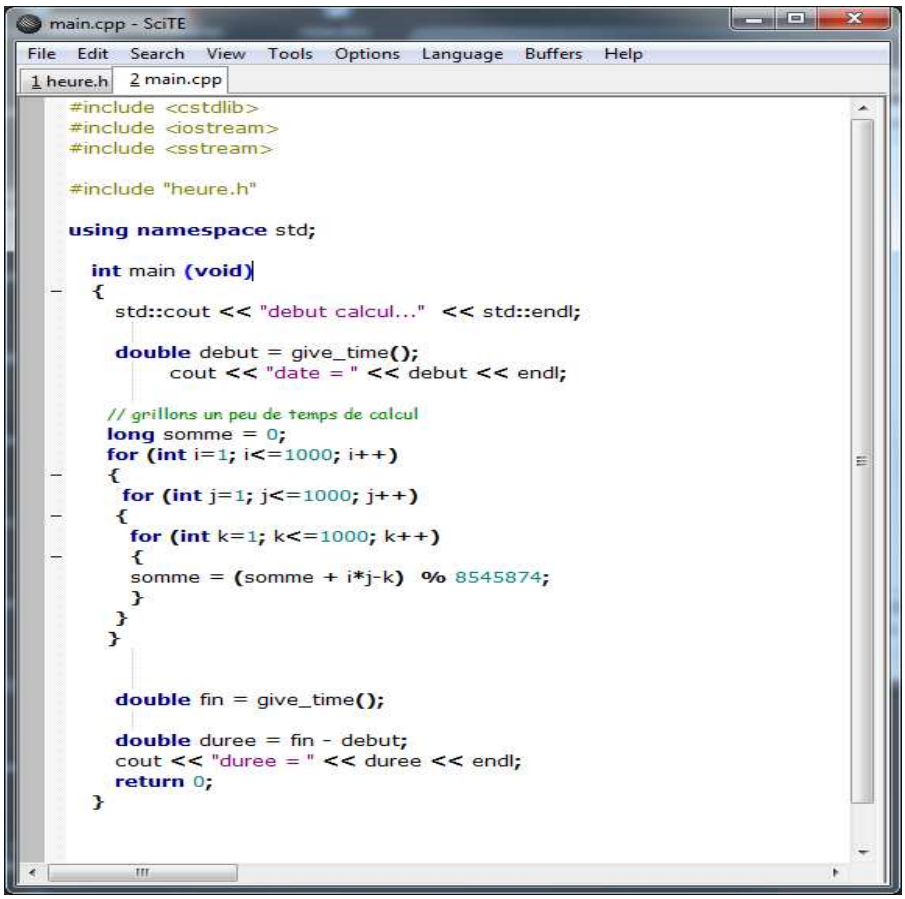

Figure 8. Le fichier « main.cpp »

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.

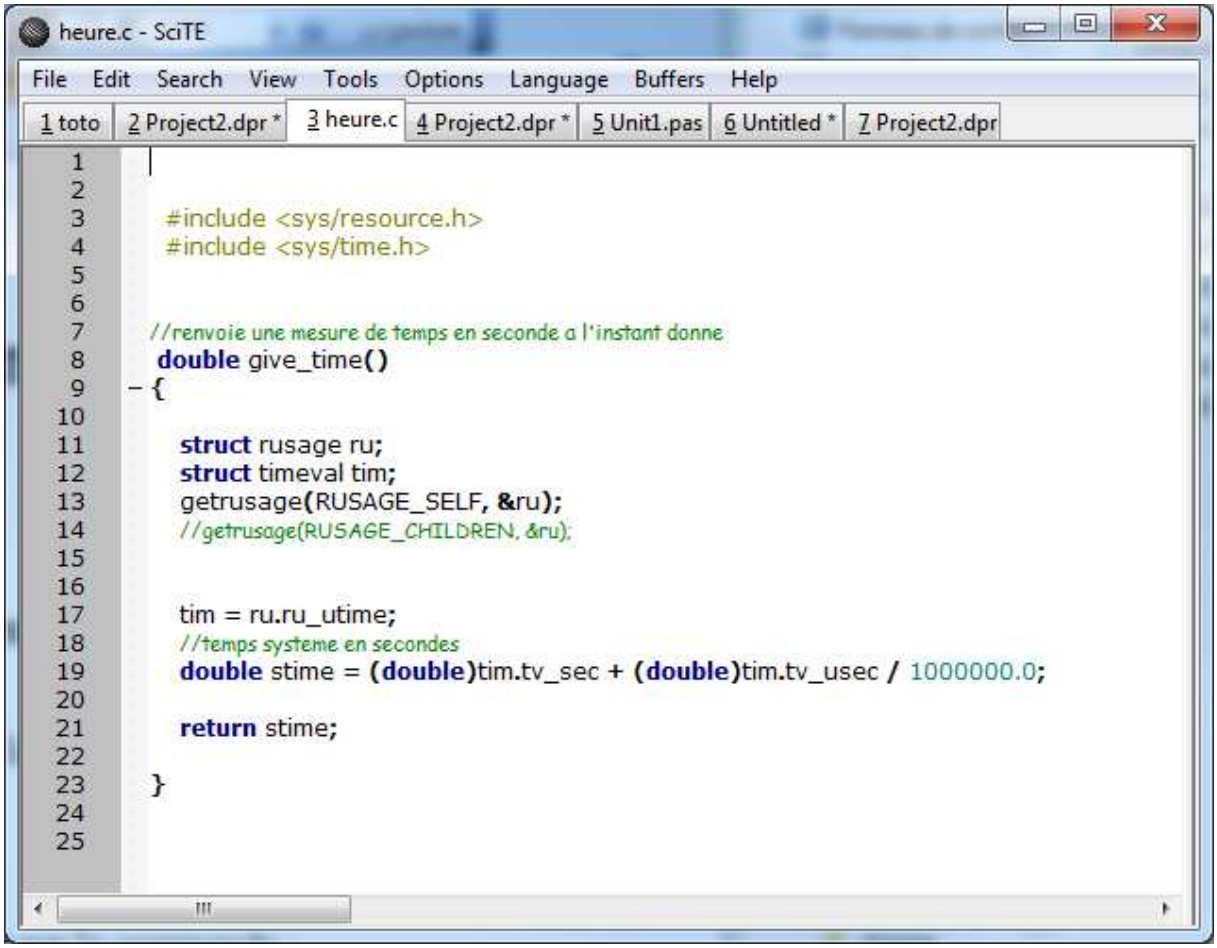

Figure 10. Le fichier « heure.c »

Sous unix, la commande est :

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

Le résultat est sur la Figure 11.

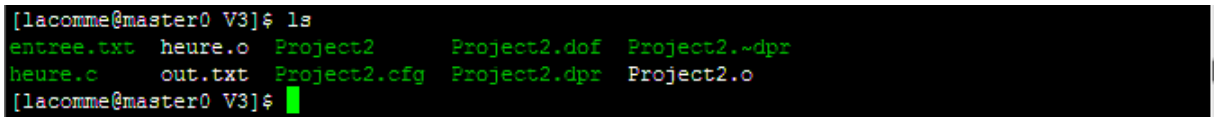

Figure 11. Création du fichier « heure.o »

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}
```

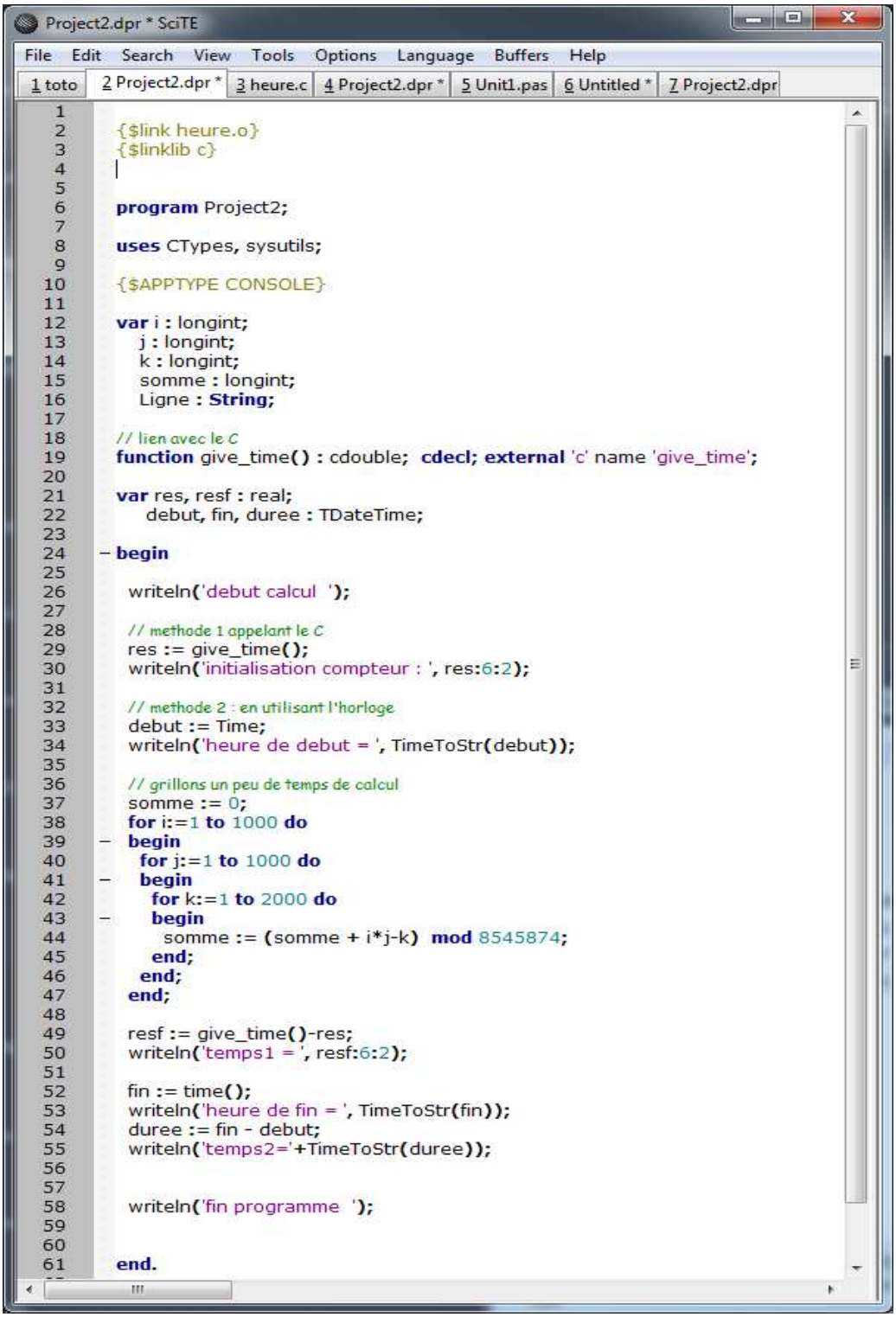

Figure 12. Le programme Free Pascal utilisant la fonction give_time écrite en C dans le fichier heure.c

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.

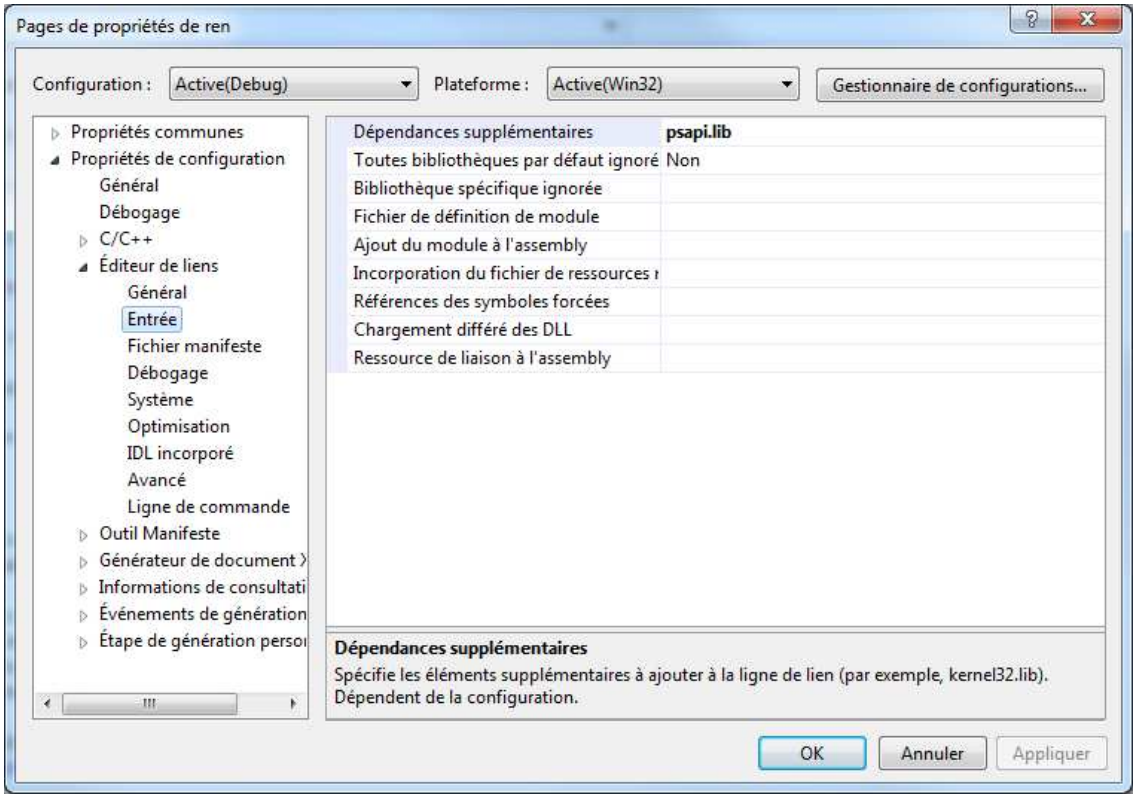

Figure 16. Configuration de l'environnement Visual C++

Le programme est très proche de la version Delphi. La seule différence est que le code est découpé en deux procédures :
- la procedure **get_date()** qui crée la liste des processus présents ;
- la procedure **print_module()** qui connaissant le numéro d'un processus parcours les processus fils à la recherche du processus à consulter (ici le programme lui-même).

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

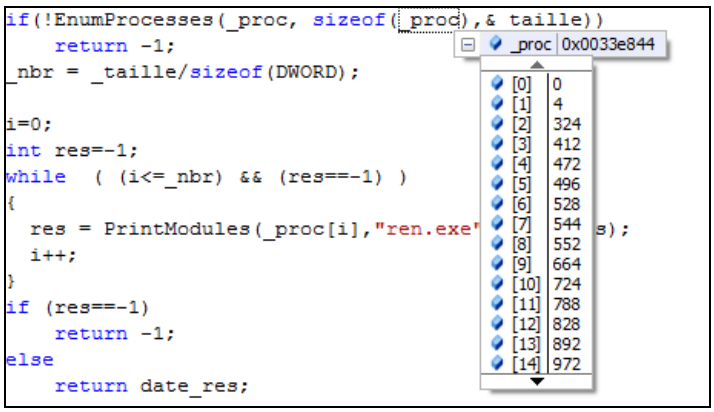

Figure 17. Tableau des PID obtenu sous Visual C++

Il suffit de parcours 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 procedure 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().**

```
int PrintModules(DWORD processID, string  str,__int64 &date_courante)
{
    HMODULE hMods[1024];
    HANDLE hProcess;
    DWORD cbNeeded;
    int resultat_comparaison_noms;
    // Get a list of all the modules in this process.
    hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE| PROCESS_TERMINATE , processID);
    if (hProcess == NULL)
       return -1;
    if (EnumProcessModules(hProcess, hMods, sizeof(hMods), &cbNeeded))
    {
        //nom du processus
        WCHAR szProcessName[512];
        if(GetModuleBaseName(hProcess,NULL,szProcessName,MAX_PATH) == 0)
        {
             return -1;
        }
        std::wstring widestr = std::wstring(str.begin(), str.end());

        resultat_comparaison_noms = wcscmp(szProcessName, widestr.c_str());

        if (resultat_comparaison_noms==0)  // on a trouve le bon
        {
            _FILETIME mCreationTime,mExitTime,mKernelTime, mUserTime;

            GetProcessTimes(hProcess,&mCreationTime,&mExitTime,&mKernelTime,&mUserTime);

            unsigned __int64 TotalTime1=((unsigned __int64)mKernelTime.dwLowDateTime | ((unsigned __int64)mKernelTime.dwHighDateTime << 32 ))
                + ((unsigned __int64)mUserTime.dwLowDateTime | ((unsigned __int64)mUserTime.dwHighDateTime << 32 ));

            // Valeur de sortie
            date_courante = (__int64)TotalTime1;

            //penser a fermer le hprocess si on ne veut pas faire cracher le programme après 8h ou 10h...
            //   ---> Je dis cela d'experiense
            CloseHandle(hProcess);
            return 1;
        }
        else
        {/* -- Rien ce n'est pas le bon ... --*/}
    }
    CloseHandle(hProcess);
    return -1;
}
```

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 methode time()) ;
- un CPU TIME de 21 (mesuré en consultant les données du processus).

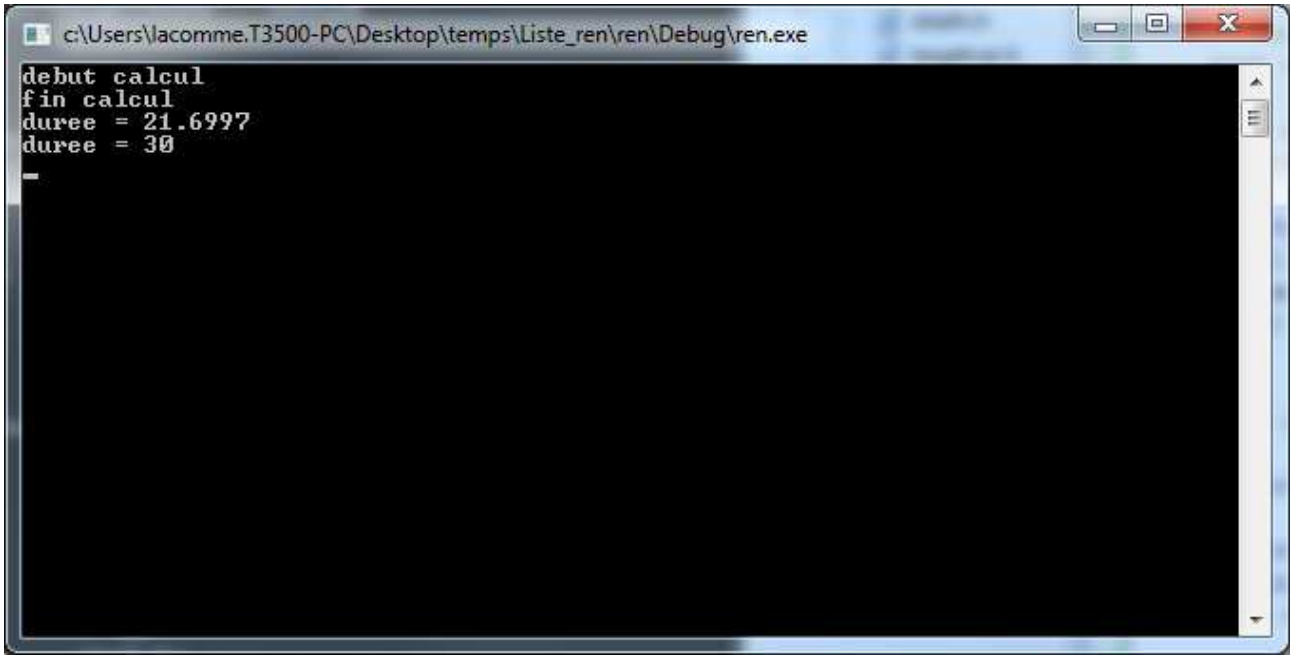

Figure 21. Test de l'application Visual C++ pendant que le processeur est utilisé à d'autre taches

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 function_parallele demarre 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;
```

Les deux structures **ThreadParam1** et **ThreadParam1** sont crées 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 crées 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.

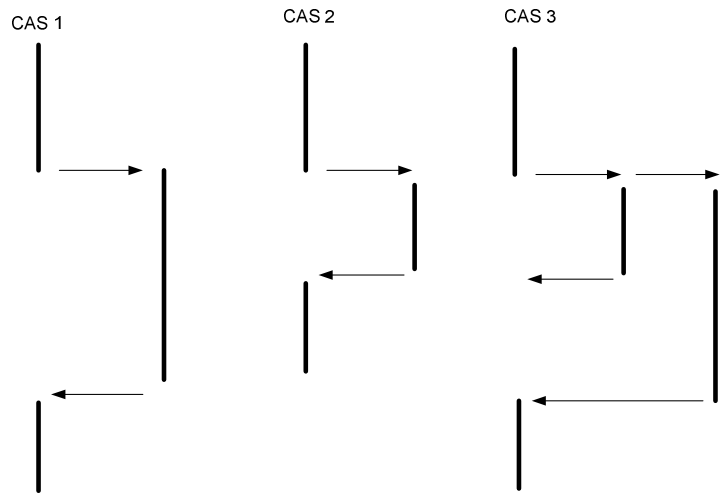

Figure 23. Les trois cas testés

Le programme utilisant directement getprocesstime sur le programme principal donne finallement 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).

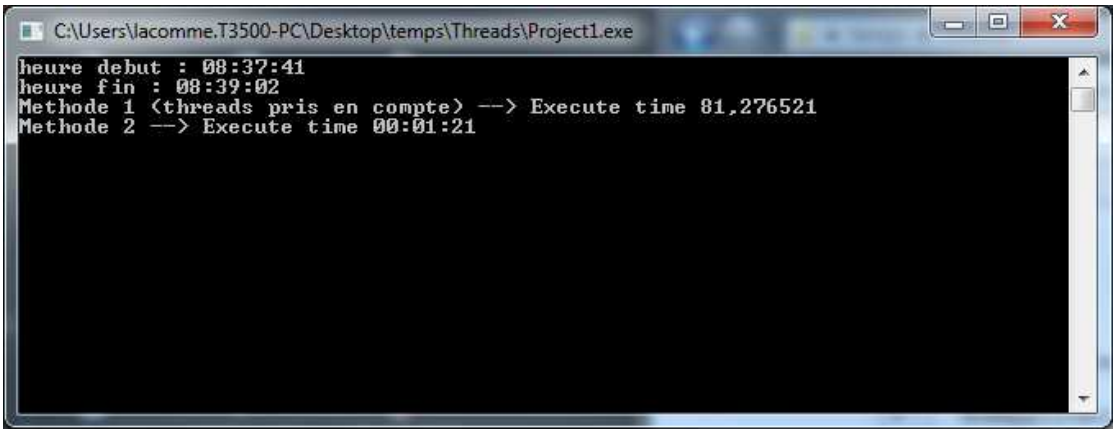

Figure 24. Résultats d'exécution du thread numéro 1 (le plus long)

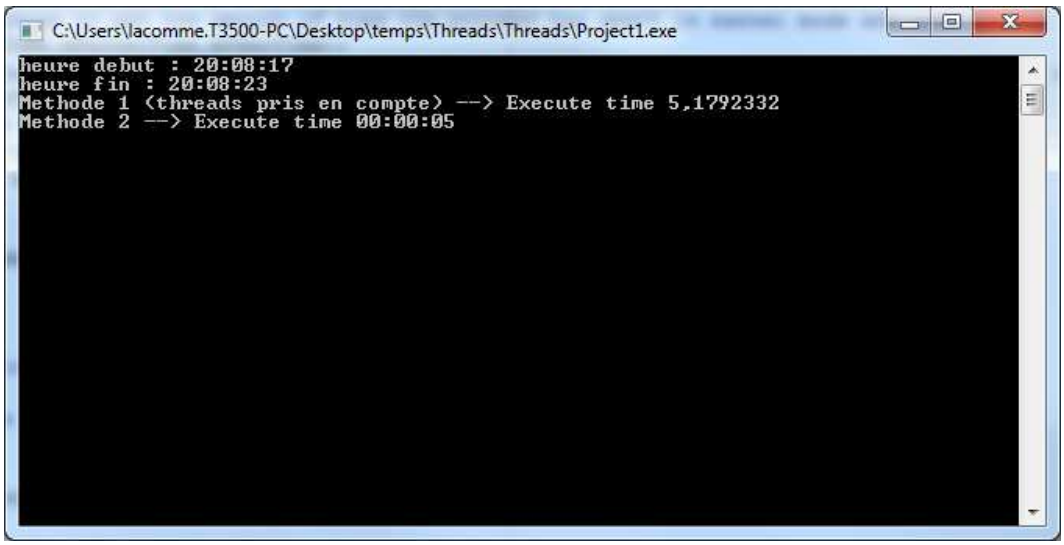

Figure 25. Résultats d'exécution du thread numéro 2 (le plus cout)

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).

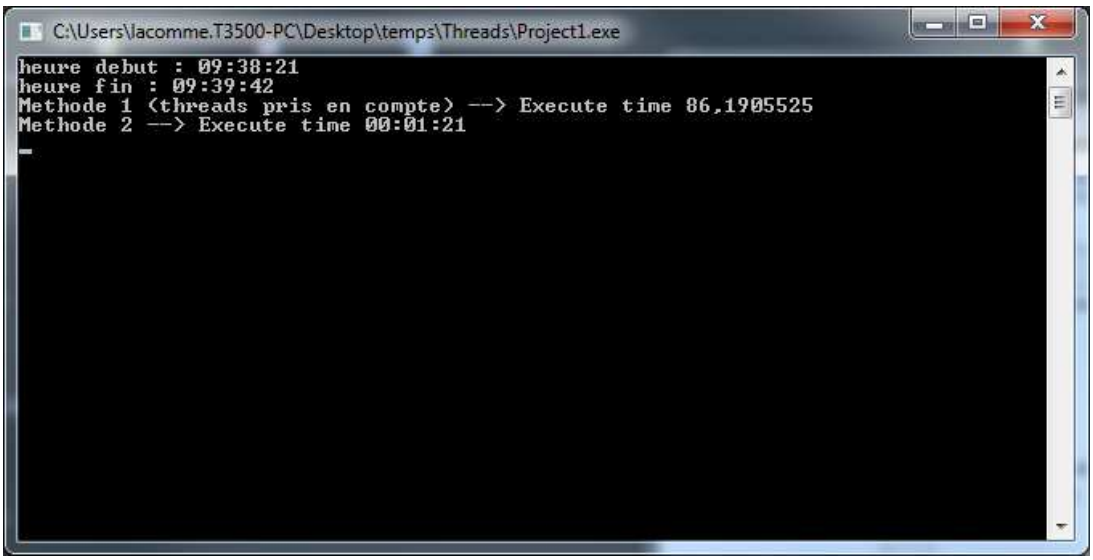

Figure 26. Résultats d'exécution avec les 2 threads

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 réaliter récuperer 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

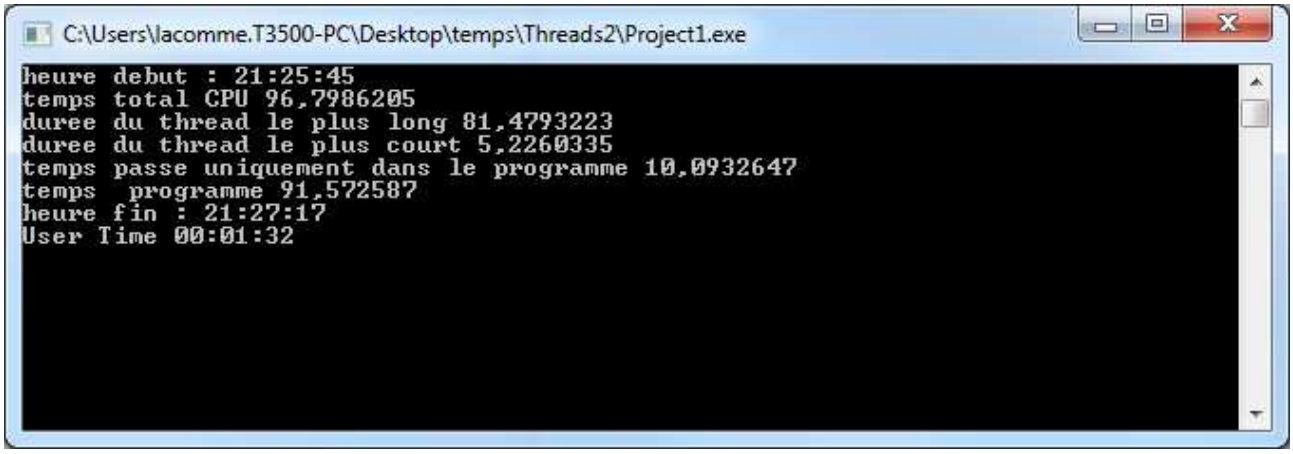

Figure 28. Mise en évidence des temps passés sur les threads