L'exécution affiche les résultats suivants :

```
$ ./exemple_setitimer_2
Temps passé en mode utilisateur : 542/100 s
Temps passé en mode noyau : 235/100 s
$ ./exemple_setitimer_2
Temps passé en mode utilisateur : 542/100 s
Temps passé en mode noyau : 240/100 s
$ ./exemple_setitimer_2
Temps passé en mode utilisateur : 554/100 s
Temps passé en mode noyau : 223/100 s
$
```

Nous voyons bien là les limites du suivi d'exécution sur un système multitâche, même si les ordres de grandeur restent bien constants. Nous copions à présent ce programme dans exemple `setitimer_3.c` en ne conservant plus que la routine de travail effectif, ce qui nous donne cette fonction `main( )` :

```
   int
main (void)
{
   action_a_mesurer( ) ;
   return (0);
}
```

Nous pouvons alors utiliser la fonction « `times` » de `bash 2`, qui permet de mesurer les temps cumulés d'exécution en mode noyau et en mode utilisateur du shell et des processus qu'il a lancés.

```
$ sh -c "./exemple_setitimer_3 ; times"
0m0.00s 0m0.00s
0m5.21s 0m2.19s
$ sh -c "./exemple_setitimer_3 ; times"
0m0.00s 0m0.01s
0m5.07s 0m2.41s
$ sh -c "./exemple_setitimer_3 ; times"
0m0.01s 0m0.00s
0m5.04s 0m2.34s
$
```

Nous voyons que les résultats sont tout à fait comparables, même s'ils présentent également une variabilité due à l'ordonnancement multitâche.

Suivre l'exécution d'un processus

Il existe plusieurs fonctions permettant de suivre l'exécution d'un processus, à la manière des routines que nous avons développées précédemment. La plus simple d'entre elles est la fonction **`clock( )`**, déclarée dans `<time.h>` ainsi :

```
clock_t clock(void);
```

Le type `clock_t` représente un temps processeur écoulé sous forme d'impulsions d'horloge *théoriques*. Nous précisons qu'il s'agit d'impulsions *théoriques* car il y a une différence d'ordre de grandeur importante entre ces quantités et la véritable horloge système utilisée par

l'ordonnanceur. À cause d'une différence entre les standards Ansi C et Posix.1, cette valeur n'a plus aucune signification effective. Pour obtenir une durée en secondes, il faut diviser la valeur `clock_t` parla constante `CLOCKS_PER_SEC`. Cette constante vaut 1 million sur l'essentiel des systèmes Unix dérivant de Système V. ainsi que sous Linux. On imagine assez bien que le séquencement des tâches est loin d'avoir effectivement lieu toutes les microsecondes...

On ne sait pas avec quelle valeur la fonction `clock( )` démarre. Il s'agit parfois de zéro. mais ce n'est pas obligatoire. Aussi est-il nécessaire de mémoriser la valeur initiale et de la sous-traire pour connaître la durée écoulée.

Sous Linux, `clock_t` est un entier long, mais ce n'est pas toujours le cas sur d'autres systèmes. Il importe donc de forcer le passage en virgule flottante pour pouvoir effectuer l'affichage. Notre programme d'exemple va mesurer le temps processeur écoulé tant en mode utilisateur qu'en mode noyau, dans la routine que nous avons déjà utilisée dans les exemples précédents.

Le programme exemple `clock.c` contient donc la fonction `main( )` suivante, en plus de la routine `action_a_mesurer( )`

exemple_clock.c

```
   int
main (void)
{
   clock_t debut_programme;
   double duree_ecoulee;
   debut_programme = clock ( ) ;
   action_a_mesurer ( );
   duree_ecoulee = clock( ) - debut_programme;
   duree_ecoulee = duree_ecoulee / CLOCKS_PER_SEC;
   fprintf (stdout, "Durée = %f \n", duree_ecoulee);
   return (0);
}
```

Les résultats sont les suivants :

```
$ ./exemple clock
Durée = 7.780000
$ ./exemple clock
Durée = 7.850000
```

Comme il fallait s'y attendre, il s'agit de la somme des temps obtenus avec nos programmes précédents, avec — comme toujours — une légère variation en fonction de la charge système.

Sous Linux, `clock_t` est équivalent à un `long int`. Il y a donc un risque de débordement de la valeur maximale au bout de `LONG_MAX` impulsions théoriques. `LONG_MAX` est une constante symbolique définie dans `<limits.h>`. Sur un PC, `LONG_MAX` vaut 2.147.483.647, et `CLOCK_PE= SEC` vaut 1.000.000. Cela donne donc une durée avant le dépassement de 2 147 secondes, soit 35 minutes.

Rappelons qu'il s'agit là de temps processeur effectif, et qu'il est assez rare qu'un programme cumule autant de temps d'exécution. Mais cela peut arriver, notamment avec des programmes