```
/*******************************************************************************
** Notes on Program Execution Time Measurements on Linux/Unix/Solaris
**
** Intended audience: Those who would like to learn more about
** measuring program execution time in modern computer systems.
**
** Used: CPE 631 Advanced Computer Systems and Architectures
**          CPE 619 Modeling and Analysis of Computer and Communication Systems
**
** ver 0.1, Spring 2007 notes on performance measurements (Solaris)
** ver 0.2, Spring 2011 updated notes on program execution time measurements (Linux)
** ver 0.3, Spring 2012 updates for execution time measurements in Linux
** ver 0.4, Spring 2018 updates for execution time measurements on Blackhawk (runs CentOS 6.9)
**
** @Aleksandar Milenkovic, milenkovic@computer.org
*******************************************************************************/
```

Program Execution Time Measurements

Let us start with a simple C program that sums up elements of an integer array.
The program initializes elements randomly and then sums the elements and prints the result.
We would like to measure program execution time.

* Copy this directory into your current directory.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-bash-4.1$ cp -r /apps/arch/arch.tut/exetimemeasure .
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

1 Using the date command

* To learn more about the date command type in man date.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-bash-4.1$ man date
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

* Inspect the code in file arrsum.c.

* Compile the code using gcc with O2 optimization level.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-bash-4.1$ gcc -O2 arrsum.c -o arrsum.exe
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

* Measure time using date command as follows.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-bash-4.1$ date && ./arrsum.exe 1024 && date
Wed Jan 24 09:04:54 CST 2018
array sum is 1114360406973
Wed Jan 24 09:04:54 CST 2018
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

In this example we get the date&time before the execution of the program (09:04:54) and after execution of the program (09:04:54). Time resolution for the date utility is 1 second, and the arrsum program takes far less time than 1 second to complete. Actually, this utility would not be useful even if we use much larger arrays. For example, if we use 1 million elements instead of 1024, we still cannot see the difference. However, if your program takes more time than 1 second, you will be able to observe differences in time and to determine program execution time.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-bash-4.1$ date && ./arrsum.exe 1000000 && date
Wed Jan 24 09:06:40 CST 2018
array sum is 1073756022677023
Wed Jan 24 09:06:40 CST 2018
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

Note: An alternative is that you invoke multiple instances of your program (e.g., 100,000 times) and measure time it takes to execute these instances as a standalone program. For example, you can have a loop that repeats your task as many times as you want. By dividing the elapsed time with the number of repetitions of your task, you can get an estimated execution time for your task.

2 Using the time command

* To learn more the time utility type in man time.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-bash-4.1$ man time
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

* A better alternative for measuring time is using time utility. The time utility prints a message with time statistics about the program run, including
(i) the elapsed real-time between the program invocation and termination
(ii) the user CPU time, and
(iii) the system CPU time.

* Examples of using time are given below:

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-bash-4.1$ time arrsum.exe 1024
array sum is 1114360406973

real    0m0.001s
user    0m0.000s
```

```
sys     0m0.000s


-bash-4.1$ time arrsum.exe 2048
array sum is 2205251413056

real    0m0.001s
user    0m0.000s
sys     0m0.000s
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

* By using the maximum number of elements we can see a change in execution time.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-bash-4.1$ time arrsum.exe 1048576
array sum is 1125894357517602

real    0m0.011s
user    0m0.010s
sys     0m0.000s
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

3 Using the clock() function

We often want to measure time execution time for a certain program section.
The clock() function allows us to do so.

* To learn more about the clock() function type in man clock.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-bash-4.1$ $ man clock
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

* Below is a typical program template for using the clock() function.

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#include <ctime.h>

....

int main(void) {

clock_t start_time, finish_time;
...

// determine overhead
start_time = clock();
finish_time = clock();
double delay_time = (double) (finish_time - start_time);
...

start_time = clock();
...// code you want to determine the execution time for
finish_time = clock();
```

```
double elapsed_time = finish_time - stat_time - delay_time;

double elapsed_time_sec = elapsed_time/CLOCKS_PER_SEC;

...
}
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

* Inspect arrsum_clock.c program that is a modified version of arrsum.c. You will see that the critical loop that sums up the elements of the array is repeated many times (constant REPEATS is set to 100,000).

* Compile the program as follows:

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-bash-4.1$ $ gcc -O2 arrsum_clock.c -o arrsum_clock.exe
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

* Run the program:

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-bash-4.1$ $ ./arrsum_clock.exe 16384
Clock ticks per second: 1000000
Loop overhead in clock ticks = 0.000000
Array sum is 1748829074928900000
Execution time for sum-up array loop in clock ticks is 6.300000
Execution time for sum-up array loop in seconds is 0.000006
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

We can conclude that loop to sum-up elements of an array with 16,384 integers takes around 6.3 us on our machine.

It is often useful for engineers to be able to quickly estimate time without using measurements like in this example. Let us try to estimate execution time for this loop. We have 16,384 iterations; let us assume that one iteration of the *for* loop requires 5 instructions (actually you can generate assembly code and see implementation of the critical loop). It is likely that critical path is created on the statement *"sum = sum + mydata[i]"* and we can reasonably assume that approximately 4 cpu clock cycles will be spent per one iteration. The entire execution time is then 16384*4cc*(1/2.93 GHz) = 22 us. The number is not far away from what we have measured.

4 Using the gettimeofday function

* To learn more about this function type in man gettimeofday.

```
<<~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

-bash-4.1$ $ man gettimeofday

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~>>
```

The function gettimeofday returns two integers.
The first one indicates the number of seconds from January 1, 1970
and the second returns the number of microseconds since the most recent second boundary.

* Below is a sample program that uses gettimeofday().

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#include <stdio.h>
#include <sys/time.h>

struct timeval start, finish ;
int msec;

int main ()
{
  gettimeofday (&start, NULL);

  sleep (200); /* wait ~ 100 seconds */

  gettimeofday (&finish, NULL);

  msec = finish.tv_sec * 1000 + finish.tv_usec / 1000;
  msec -= start.tv_sec * 1000 + start.tv_usec / 1000;

  printf("Time: %d milliseconds\n", msec);
}
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

* This utility is useful for measuring execution times of tasks that take a lot of time to complete.
The clock() function overflows depending on the length of clock_t. For example, if CLOCKS_PER_SEC is 1,000,000 and clock_t is 32-bit, the overflow will occur every 2^32/1,000,000 seconds, which is approximately every 72 hours.

5 Using PAPI

See the LaCASA Getting Started with PAPI page at
http://lacasa.uah.edu/portal/index.php/tutorials/34-getting-started-with-papi and PAPI documentation.

6 Using perf

See the LaCASA PerfTool tutorial at http://lacasa.uah.edu/portal/index.php/tutorials/28-perf-tool and perf documentation.